# CS336 Assignment 1 (basics): Building a Transformer LM

Version 26.0.3

CS336 Staff

Spring 2026

## 1 Assignment Overview

In this assignment, you will build all the components needed to train a standard Transformer language model (LM) from scratch and train some models.

### What you will implement

1. Byte-pair encoding (BPE) tokenizer (Section 2)
2. Transformer language model (LM) (Section 3)
3. The cross-entropy loss function and the AdamW optimizer (Section 4)
4. The training loop, with support for serializing and loading model and optimizer state (Section 5)

### What you will run

1. BPE tokenizer training on the TinyStories dataset.
2. Trained tokenizer encoding on the dataset to convert it into a sequence of integer IDs.
3. Transformer LM training on the TinyStories dataset.
4. Sample generation and evaluation of perplexity using the trained Transformer LM.
5. Model training on OpenWebText, and submit your attained perplexities to a leaderboard.

### What you can use

We expect you to build each component from scratch. In particular, you may *not* use any definitions from `torch.nn`, `torch.nn.functional`, or `torch.optim` except for the following:

- `torch.nn.Parameter`
- Container classes in `torch.nn` (e.g., `Module`, `ModuleList`, `Sequential`, etc.).[1]
- The `torch.optim.Optimizer` base class

You may use any other PyTorch definitions. If you would like to use a function or class and are not sure whether it is permitted, feel free to ask on Slack. When in doubt, consider if using it compromises the "from-scratch" ethos of the assignment.

### Statement on AI tools

AI can solve many parts of the assignments fully autonomously. This makes it harder to deeply engage with, and learn from, the course material.

The use of AI tools is permitted for answering high-level conceptual questions, or for providing low-level programming documentation like function signatures and library APIs. However, AI tools are not permitted for implementing any part of any assignment. This includes both coding agents (e.g., Cursor Agents, Codex, Claude Code) and AI autocomplete (e.g., Cursor Tab, GitHub Copilot). When using an AI agent, make sure it uses the AGENTS.md file provided. The prompt should also be included when using chatbots.

[1] See pytorch.org/docs/stable/nn.html#containers for a full list.

We strongly encourage you to disable AI autocomplete (e.g., Cursor Tab, GitHub Copilot) in your IDE when completing assignments (though non-AI autocomplete, e.g., autocompleting function names is totally fine). Previous students have highlighted that disabling AI autocomplete made it easier to engage deeply with the material.

For the full AI Policy, see this document.

### What the code looks like

The assignment code, as well as this writeup, are available on GitHub at:

`github.com/stanford-cs336/assignment1-basics`

Please `git clone` the repository. If there are any updates, we will notify you so you can `git pull` to get the latest.

1. `cs336_basics/*`: This is where you write your code. Note that there's no code in here—you can do whatever you want from scratch!
2. `adapters.py`: There is a set of functionality that your code must have. For each piece of functionality (e.g., scaled dot product attention), fill out its implementation (e.g., `run_scaled_dot_product_attention`) by simply invoking your code. Note: your changes to `adapters.py` should not contain any substantive logic; this is glue code.
3. `test_*.py`: This contains all the tests that you must pass (e.g., `test_scaled_dot_product_attention`), which will invoke the hooks defined in `adapters.py`. Don't edit the test files.

### How to submit

In order to submit, run `make_submission.sh` to construct a submission zip file. Make sure to add any additional files to the list of exclusions in the script if you have large data files or checkpoints you don't want to include in your submission zip.

You will submit the following files to Gradescope:

- `writeup.pdf`: Answer all the written questions. Please typeset your responses.
- `code.zip`: Contains all the code you've written.

To submit to the leaderboard, submit a PR to:

`github.com/stanford-cs336/assignment1-basics-leaderboard`

See the `README.md` in the leaderboard repository for detailed submission instructions.

### Where to get datasets

This assignment will use two pre-processed datasets: TinyStories [R. Eldan et al., 2023] and OpenWebText [A. Gokaslan et al., 2019]. Both datasets are single, large plaintext files.

If you are doing the assignment with the class, you can find instructions for downloading the dataset in the compute guide.

If you are following along at home, you can download these files with the commands inside the `README.md`.

**Low-Resource Tip: Init**

Throughout the course's assignment handouts, we will give advice for working through parts of the assignment with fewer or no GPU resources. For example, we will sometimes suggest **downscaling** your dataset or model size, or explain how to run training code on a Mac

integrated GPU or CPU. You'll find these "low-resource tips" in a blue box (like this one). Even if you are an enrolled Stanford student with access to the course machines, these tips may help you iterate faster and save time, so we recommend reading them!

**Low-Resource Tip: Assignment 1 on Apple Silicon or CPU**

With the staff solution code, we can train an LM to generate reasonably fluent text on an Apple M4 Max chip with 36 GB RAM, in under 5 minutes on Metal GPU (MPS) and about 30 minutes using the CPU. If these words don't mean much to you, don't worry! Just know that if you have a reasonably up-to-date laptop and your implementation is correct and efficient, you will be able to train a small LM that generates simple children's stories with decent fluency.

Later in the assignment, we will explain what changes to make if you are on CPU or MPS.

# 2 Byte-Pair Encoding (BPE) Tokenizer

In the first part of the assignment, we will train and implement a byte-level byte-pair encoding (BPE) tokenizer [R. Sennrich et al., 2016; C. Wang et al., 2019]. In particular, we will represent arbitrary (Unicode) strings as a sequence of bytes and train our BPE tokenizer on this byte sequence. Later, we will use this tokenizer to encode text (a string) into tokens (a sequence of integers) for language modeling.

## 2.1 The Unicode Standard

Unicode is a text encoding standard that maps characters to integer *code points*. As of Unicode 17.0 (released in September 2025), the standard defines 159,801 characters across 172 scripts. For example, the character "s" has the code point 115 (typically notated as `U+0073`, where `U+` is a conventional prefix and `0073` is 115 in hexadecimal), and the character "牛" has the code point 29275. In Python, you can use the `ord()` function to convert a single Unicode character into its integer representation. The `chr()` function converts an integer Unicode code point into a string with the corresponding character.

```
>>> ord('牛')
29275
>>> chr(29275)
'牛'
```

**Problem (`unicode1`): Understanding Unicode (1 point)**

(a) What Unicode character does `chr(0)` return?

**Deliverable**: A one-sentence response.

(b) How does this character's string representation (`__repr__()`) differ from its printed representation?

**Deliverable**: A one-sentence response.

(c) What happens when this character occurs in text? It may be helpful to play around with the following in your Python interpreter and see if it matches your expectations:

```
>>> chr(0)
>>> print(chr(0))
```

```
>>> "this is a test" + chr(0) + "string"
>>> print("this is a test" + chr(0) + "string")
```

**Deliverable**: A one-sentence response.

## 2.2 Unicode Encodings

While the Unicode standard defines a mapping from characters to code points (integers), it's impractical to train tokenizers directly on Unicode code points, since the vocabulary would be prohibitively large (around 150K items) and sparse (since many characters are quite rare). Instead, we'll use a Unicode encoding, which converts a Unicode character into a sequence of bytes. The Unicode standard itself defines three encodings: UTF-8, UTF-16, and UTF-32, with UTF-8 being the dominant encoding for the Internet (more than 98% of all webpages).

To encode a Unicode string into UTF-8, we can use the `encode()` function in Python. To access the underlying byte values for a Python `bytes` object, we can iterate over it (e.g., call `list()`). Finally, we can use the `decode()` function to decode a UTF-8 byte string into a Unicode string.

```
>>> test_string = "hello! こんにちは!"
>>> utf8_encoded = test_string.encode("utf-8")
>>> print(utf8_encoded)
b'hello! \xe3\x81\x93\xe3\x82\x93\xe3\x81\xab\xe3\x81\xa1\xe3\x81\xaf!'
>>> print(type(utf8_encoded))
<class 'bytes'>
>>> # Get the byte values for the encoded string (integers from 0 to 255).
>>> list(utf8_encoded)
[104, 101, 108, 108, 111, 33, 32, 227, 129, 147, 227, 130, 147, 227, 129, 171, 227, 129,
161, 227, 129, 175, 33]
>>> # One byte does not necessarily correspond to one Unicode character!
>>> print(len(test_string))
13
>>> print(len(utf8_encoded))
23
>>> print(utf8_encoded.decode("utf-8"))
hello! こんにちは!
```

By converting our Unicode code points into a sequence of bytes (e.g., via the UTF-8 encoding), we are essentially taking a sequence of code points (21-bit integers with 159,801 valid values) and transforming it into a sequence of byte values (integers in the range 0 to 255). The 256-length byte vocabulary is *much* more manageable to deal with. When using byte-level tokenization, we do not need to worry about out-of-vocabulary tokens, since we know that *any* input text can be expressed as a sequence of integers from 0 to 255.

**Problem (`unicode2`): Unicode Encodings (3 points)**

(a) What are some reasons to prefer training our tokenizer on UTF-8 encoded bytes, rather than UTF-16 or UTF-32? It may be helpful to compare the output of these encodings for various input strings.

**Deliverable**: A one-to-two sentence response.

(b) Consider the following (incorrect) function, which is intended to decode a UTF-8 byte string into a Unicode string. Why is this function incorrect? Provide an example of an input byte string that yields incorrect results.

```
def decode_utf8_bytes_to_str_wrong(bytestring: bytes):
    return "".join([bytes([b]).decode("utf-8") for b in bytestring])

>>> decode_utf8_bytes_to_str_wrong("hello".encode("utf-8"))
'hello'
```

**Deliverable**: An example input byte string for which `decode_utf8_bytes_to_str_wrong` produces incorrect output, with a one-sentence explanation of why the function is incorrect.

(c) Give a two-byte sequence that does not decode to any Unicode character(s).

**Deliverable**: An example, with a one-sentence explanation.

## 2.3 Subword Tokenization

While byte-level tokenization can alleviate the out-of-vocabulary issues faced by word-level tokenizers, tokenizing text into bytes results in extremely long input sequences. This slows down model training, since a sentence with 10 words might only be 10 tokens long in a word-level language model, but could be 50 or more tokens long in a character-level model (depending on the length of the words). Processing these longer sequences requires more computation at each step of the model. Furthermore, language modeling on byte sequences is difficult because the longer input sequences create long-term dependencies in the data.

Subword tokenization is a midpoint between word-level tokenizers and byte-level tokenizers. Note that a byte-level tokenizer's vocabulary has 256 entries (byte values are 0 to 255). A subword tokenizer trades off a larger vocabulary size for better compression of the input byte sequence. For example, if the byte sequence `b'the'` often occurs in our raw text training data, assigning it an entry in the vocabulary would reduce this 3-token sequence to a single token.

How do we select these subword units to add to our vocabulary? R. Sennrich et al. [3] propose to use byte-pair encoding (BPE; P. Gage [5]), a compression algorithm that iteratively replaces ("merges") the most frequent pair of bytes with a single, new unused index. Note that this algorithm adds subword tokens to our vocabulary to maximize the compression of our input sequences—if a word occurs in our input text enough times, it'll be represented as a single subword unit.

Subword tokenizers with vocabularies constructed via BPE are often called BPE tokenizers. In this assignment, we'll implement a byte-level BPE tokenizer, where the vocabulary items are bytes or merged sequences of bytes, which give us the best of both worlds in terms of out-of-vocabulary handling and manageable input sequence lengths. The process of constructing the BPE tokenizer vocabulary is known as "training" the BPE tokenizer.

## 2.4 BPE Tokenizer Training

The BPE tokenizer training procedure consists of three main steps.

### Vocabulary initialization

The tokenizer vocabulary is a one-to-one mapping from bytestring token to integer ID. Since we're training a byte-level BPE tokenizer, our initial vocabulary is simply the set of all bytes. Since there are 256 possible byte values, our initial vocabulary is of size 256.

### Pre-tokenization

Once you have a vocabulary, you could, in principle, count how often bytes occur next to each other in your text and begin merging them starting with the most frequent pair of bytes. However, this is quite computationally expensive, since we'd have to take a full pass over the corpus each time we merge. In

addition, directly merging bytes across the corpus may result in tokens that differ only in punctuation (e.g., `dog!` vs. `dog.`). These tokens would get completely different token IDs, even though they are likely to have high semantic similarity (since they differ only in punctuation).

To avoid this, we *pre-tokenize* the corpus. You can think of this as a coarse-grained tokenization over the corpus that helps us count how often pairs of characters appear. For example, the word `'text'` might be a pre-token that appears 10 times. In this case, when we count how often the characters 't' and 'e' appear next to each other, we will see that the word 'text' has 't' and 'e' adjacent and we can increment their count by 10 instead of looking through the corpus. Since we're training a byte-level BPE model, each pre-token is represented as a sequence of UTF-8 bytes.

The original BPE implementation of R. Sennrich et al. [3] pre-tokenizes by simply splitting on whitespace (i.e., `s.split(" ")`). This method is still found in tokenizers based on SentencePiece (for instance the Llama 1 and 2 tokenizer).

Most modern tokenizers use a regex-based pre-tokenizer, a practice from GPT-2; A. Radford et al. [6]. We'll use a slightly prettier form of the original regex, fetched from `github.com/openai/tiktoken/pull/234/files`:

```
>>> PAT = r"""'(?:[sdmt]|ll|ve|re)| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+(?!\S)|\s+"""
```

It may be useful to interactively split some text with this pre-tokenizer to get a better sense of its behavior:

```
>>> # requires `regex` package
>>> import regex as re
>>> re.findall(PAT, "some text that i'll pre-tokenize")
['some', ' text', ' that', ' i', "'ll", ' pre', '-', 'tokenize']
```

When using it in your code, however, you should use `re.finditer` to avoid storing the pre-tokenized words as you construct your mapping from pre-tokens to their counts.

### Compute BPE merges

Now that we've converted our input text into pre-tokens and represented each pre-token as a sequence of UTF-8 bytes, we can compute the BPE merges (i.e., train the BPE tokenizer). At a high level, the BPE algorithm iteratively counts every pair of bytes and identifies the pair with the highest frequency ("A", "B"). Every occurrence of this most frequent pair ("A", "B") is then *merged*, i.e., replaced with a new token "AB". This new merged token is added to our vocabulary; as a result, the final vocabulary after BPE training is the size of the initial vocabulary (256 in our case), plus the number of BPE merge operations performed during training. For efficiency during BPE training, we do not consider pairs that cross pre-token boundaries.[2] When computing merges, deterministically break ties in pair frequency by *preferring the lexicographically greater pair*. For example, if the pairs ("A", "B"), ("A", "C"), ("B", "ZZ"), and ("BA", "A") all have the highest frequency, we'd merge ("BA", "A"):

```
>>> max([("A", "B"), ("A", "C"), ("B", "ZZ"), ("BA", "A")])
('BA', 'A')
```

### Special tokens

Often, some strings (e.g., `<|endoftext|>`) are used to encode metadata (e.g., boundaries between documents). When encoding text, it's often desirable to treat some strings as "special tokens" that should

[2]Note that the original BPE formulation R. Sennrich et al. [3] specifies the inclusion of an end-of-word token. We do not add an end-of-word-token when training byte-level BPE models because all bytes (including whitespace and punctuation) are included in the model's vocabulary. Since we're explicitly representing spaces and punctuation, the learned BPE merges will naturally reflect these word boundaries.

never be split into multiple tokens (i.e., will always be preserved as a single token). For example, the end-of-sequence string `<|endoftext|>` should always be preserved as a single token (i.e., a single integer ID), so we know when to stop generating from the language model. These special tokens must be added to the vocabulary, so they have a corresponding fixed token ID.

Algorithm 1 of R. Sennrich et al. [3] contains an inefficient implementation of BPE tokenizer training (essentially following the steps that we outlined above). As a first exercise, it may be useful to implement and test this function to check your understanding.

**Example (`bpe_example`): BPE training example**

---

Here is a stylized example from R. Sennrich et al. [3]. Consider a corpus consisting of the following text

```
low low low low low
lower lower widest widest widest
newest newest newest newest newest newest
```

and the vocabulary has a special token `<|endoftext|>`.

**Vocabulary**

We initialize our vocabulary with our special token `<|endoftext|>` and the 256 byte values.

**Pre-tokenization**

For simplicity and to focus on the merge procedure, we assume in this example that pre-tokenization simply splits on whitespace. When we pre-tokenize and count, we end up with the frequency table.

```
{low: 5, lower: 2, widest: 3, newest: 6}
```

It is convenient to represent this as a `dict[tuple[bytes, ...], int]`, e.g. {`(l,o,w)`: 5, …}. Note that even a single byte is a `bytes` object in Python. There is no `byte` type in Python to represent a single byte, just as there is no `char` type in Python to represent a single character.

**Merges**

We first look at every successive pair of bytes and sum the frequency of the words where they appear {`lo`: 7, `ow`: 7, `we`: 8, `er`: 2, `wi`: 3, `id`: 3, `de`: 3, `es`: 9, `st`: 9, `ne`: 6, `ew`: 6}. The pairs `('e', 's')` and `('s', 't')` are tied, so we take the lexicographically greater pair, `('s', 't')`. We would then merge the pre-tokens so that we end up with {`(l,o,w)`: 5, `(l,o,w,e,r)`: 2, `(w,i,d,e,st)`: 3, `(n,e,w,e,st)`: 6}.

In the second round, we see that `(e, st)` is the most common pair (with a count of 9) and we would merge into {`(l,o,w)`: 5, `(l,o,w,e,r)`: 2, `(w,i,d,est)`: 3, `(n,e,w,est)`: 6}. Continuing this, the sequence of merges we get in the end will be `['s t', 'e st', 'o w', 'l ow', 'w est', 'n e', 'ne west', 'w i', 'wi d', 'wid est', 'low e', 'lowe r']`.

If we take 6 merges, we have `['s t', 'e st', 'o w', 'l ow', 'w est', 'n e']` and our vocabulary elements would be `[<|endoftext|>, [...256 BYTE CHARS], st, est, ow, low, west, ne]`.

With this vocabulary and set of merges, the word `newest` would tokenize as `[ne, west]`.

## 2.5 Experimenting with BPE Tokenizer Training

Let's train a byte-level BPE tokenizer on the TinyStories dataset. Instructions to find / download the dataset can be found in Section 1. Before you start, we recommend taking a look at the TinyStories dataset to get a sense of what's in the data.

**Parallelizing pre-tokenization**

You will find that a major bottleneck is the pre-tokenization step. You can speed up pre-tokenization by parallelizing your code with the built-in library multiprocessing. Concretely, we recommend that in parallel implementations of pre-tokenization, you chunk the corpus while ensuring your chunk boundaries occur at the beginning of a special token. You are free to use the starter code at the following link verbatim to obtain chunk boundaries, which you can then use to distribute work across your processes:

https://github.com/stanford-cs336/assignment1-basics/blob/main/cs336_basics/pretokenization_example.py

This chunking will always be valid, since we never want to merge across document boundaries. For the purposes of the assignment, you can always split in this way. Don't worry about the edge case of receiving a very large corpus that does not contain <|endoftext|>.

**Removing special tokens before pre-tokenization**

Before running pre-tokenization with the regex pattern (using re.finditer), you should strip out all special tokens from your corpus (or your chunk, if using a parallel implementation). Make sure that you **split** on your special tokens, so that no merging can occur across the text they delimit. For example, if you have a corpus (or chunk) like [Doc 1]<|endoftext|>[Doc 2], you should split on the special token <|endoftext|>, and pre-tokenize [Doc 1] and [Doc 2] separately, so that no merging can occur across the document boundary. In other words, special tokens define hard segmentation boundaries during training, but they should not themselves contribute to merge counts. This can be done using re.split with "|".join(special_tokens) as the delimiter (with careful use of re.escape since | may occur in the special tokens). The test test_train_bpe_special_tokens will test for this.

**Optimizing the merging step**

The naïve implementation of BPE training in the stylized example above is slow because for every merge, it iterates over all byte pairs to identify the most frequent pair. However, the only pair counts that change after each merge are those that overlap with the merged pair. Thus, BPE training speed can be improved by indexing the counts of all pairs and incrementally updating these counts, rather than explicitly iterating over each pair of bytes to count pair frequencies. You can get significant speedups with this caching procedure, though we note that the merging part of BPE training is *not* parallelizable in Python.

**Low-Resource Tip: Profiling**

---

You should use profiling tools like cProfile or py-spy to identify the bottlenecks in your implementation, and focus on optimizing those.

**Low-Resource Tip: "Downscaling"**

---

Instead of jumping to training your tokenizer on the full TinyStories dataset, we recommend you first train on a small subset of the data: a "debug dataset". For example, you could train your tokenizer on the TinyStories validation set instead, which is 22K documents instead of 2.12M. This illustrates a general strategy of downscaling whenever possible to speed up development: for

example, using smaller datasets, smaller model sizes, etc. Choosing the size of the debug dataset or hyperparameter config requires careful consideration: you want your debug set to be large enough to have the same bottlenecks as the full configuration (so that the optimizations you make will generalize), but not so big that it takes forever to run.

**Problem (`train_bpe`): BPE Tokenizer Training (15 points)**

---

**Deliverable**: Write a function that, given a path to an input text file, trains a (byte-level) BPE tokenizer. Your BPE training function should handle (at least) the following input parameters:

**Input**

**`input_path: str`** Path to a text file with BPE tokenizer training data.

**`vocab_size: int`** A positive integer that defines the maximum final vocabulary size (including the initial byte vocabulary, vocabulary items produced from merging, and any special tokens).

**`special_tokens: list[str]`** A list of strings to add to the vocabulary. During training, treat them as hard boundaries that prevent merges across their spans, but do not include them when computing merge statistics.

Your BPE training function should return the resulting vocabulary and merges:

**Output**

**`vocab: dict[int, bytes]`** The tokenizer vocabulary, a mapping from `int` (token ID in the vocabulary) to `bytes` (token bytes).

**`merges: list[tuple[bytes, bytes]]`** A list of BPE merges produced from training. Each list item is a `tuple` of `bytes` (`<token1>, <token2>`), representing that `<token1>` was merged with `<token2>`. The merges should be ordered by order of creation.

To test your BPE training function against our provided tests, you will first need to implement the test adapter at [`adapters.run_train_bpe`]. Then, run `uv run pytest tests/test_train_bpe.py`. Your implementation should be able to pass all tests. Optionally (this could be a large time-investment), you can implement the key parts of your training method using some systems language, for instance C++ (consider `cppyy` or `nanobind`) or Rust (using PyO3). If you do this, be aware of which operations require copying vs reading directly from Python memory, and make sure to leave build instructions, or make sure it builds using only `pyproject.toml`. Also note that the GPT-2 regex is not well-supported in most regex engines and will be too slow in most that do. We have verified that Oniguruma is reasonably fast and supports negative lookahead, but the `regex` package in Python is, if anything, even faster.

**Problem (`train_bpe_tinystories`): BPE Training on TinyStories (2 points)**

---

(a) Train a byte-level BPE tokenizer on the TinyStories dataset, using a maximum vocabulary size of 10,000. Make sure to add the TinyStories `<|endoftext|>` special token to the vocabulary. Serialize the resulting vocabulary and merges to disk for further inspection. How much time and memory did training take? What is the longest token in the vocabulary? Does it make sense?

**Resource requirements**: $\leq$ 30 minutes (no GPUs), $\leq$ 30 GB RAM

**Hint** You should be able to get under 2 minutes for BPE training using `multiprocessing` during pre-tokenization and the following two facts:

(a) The <|endoftext|> token delimits documents in the data files.

(b) The <|endoftext|> token is handled as a special case before the BPE merges are applied.

**Deliverable**: A one-to-two sentence response.

(b) Profile your code. What part of the tokenizer training process takes the most time?

**Deliverable**: A one-to-two sentence response.

Next, we'll try training a byte-level BPE tokenizer on the OpenWebText dataset. As before, we recommend taking a look at the dataset to better understand its contents.

**Problem (`train_bpe_expts_owt`): BPE Training on OpenWebText (2 points)**

---

(a) Train a byte-level BPE tokenizer on the OpenWebText dataset, using a maximum vocabulary size of 32,000. Serialize the resulting vocabulary and merges to disk for further inspection. What is the longest token in the vocabulary? Does it make sense?

**Resource requirements**: $\leq$ 12 hours (no GPUs), $\leq$ 100 GB RAM

**Deliverable**: A one-to-two sentence response.

(b) Compare and contrast the tokenizer that you get training on TinyStories versus OpenWebText.

**Deliverable**: A one-to-two sentence response.

## 2.6 BPE Tokenizer: Encoding and Decoding

In the previous part of the assignment, we implemented a function to train a BPE tokenizer on input text to obtain a tokenizer vocabulary and a list of BPE merges. Now, we will implement a BPE tokenizer that loads a provided vocabulary and list of merges and uses them to encode and decode text to/from token IDs.

### 2.6.1 Encoding text

The process of encoding text by BPE mirrors how we train the BPE vocabulary. There are a few major steps.

**Step 1: Pre-tokenize.** We first pre-tokenize the sequence and represent each pre-token as a sequence of UTF-8 bytes, just as we did in BPE training. We will be merging these bytes within each pre-token into vocabulary elements, handling each pre-token independently (no merges across pre-token boundaries).

**Step 2: Apply the merges.** We then take the sequence of vocabulary element merges created during BPE training, and apply it to our pre-tokens *in the same order of creation.*

**Example (`bpe_encoding`): BPE encoding example**

---

For example, suppose our input string is `'the cat ate'`, our vocabulary is `{0: b' ', 1: b'a', 2: b'c', 3: b'e', 4: b'h', 5: b't', 6: b'th', 7: b' c', 8: b' a', 9: b'the', 10: b' at'}`, and our learned merges are `[(b't', b'h'), (b' ', b'c'), (b' ', b'a'), (b'th', b'e'), (b' a',`

b't')]. First, our pre-tokenizer would split this string into ['the', ' cat', ' ate']. Then, we'll look at each pre-token and apply the BPE merges.

The first pre-token 'the' is initially represented as [b't', b'h', b'e']. Looking at our list of merges, we identify the first applicable merge to be (b't', b'h'), and use that to transform the pre-token into [b'th', b'e']. Then, we go back to the list of merges and identify the next applicable merge to be (b'th', b'e'), which transforms the pre-token into [b'the']. Finally, looking back at the list of merges, we see that there are no more that apply to the string (since the entire pre-token has been merged into a single token), so we are done applying the BPE merges. The corresponding integer sequence is [9].

Repeating this process for the remaining pre-tokens, we see that the pre-token ' cat' is represented as [b' c', b'a', b't'] after applying the BPE merges, which becomes the integer sequence [7, 1, 5]. The final pre-token ' ate' is [b' at', b'e'] after applying the BPE merges, which becomes the integer sequence [10, 3]. Thus, the final result of encoding our input string is [9, 7, 1, 5, 10, 3].

**Special tokens**

Your tokenizer should be able to properly handle user-defined special tokens when encoding text (provided when constructing the tokenizer).

**Memory considerations**

Suppose we want to tokenize a large text file that we cannot fit in memory. To efficiently tokenize this large file (or any other stream of data), we need to break it up into manageable chunks and process each chunk in turn, so that the memory complexity is constant as opposed to linear in the size of the text. In doing so, we need to make sure that a token doesn't cross chunk boundaries, else we'll get a different tokenization than the naïve method of tokenizing the entire sequence in-memory.

### 2.6.2 Decoding text

To decode a sequence of integer token IDs back to raw text, we can simply look up each ID's corresponding entries in the vocabulary (a byte sequence), concatenate them together, and then decode the bytes to a Unicode string. Note that input IDs are not guaranteed to map to valid Unicode strings (since a user could input any sequence of integer IDs). In the case that the input token IDs do not produce a valid Unicode string, you should replace the malformed bytes with the official Unicode replacement character U+FFFD.[3] The errors argument of bytes.decode controls how Unicode decoding errors are handled, and using errors='replace' will automatically replace malformed data with the replacement marker.

**Problem (tokenizer): Implementing the tokenizer (15 points)**

**Deliverable**: Implement a Tokenizer class that, given a vocabulary and a list of merges, encodes text into integer IDs and decodes integer IDs into text. Your tokenizer should also support user-provided special tokens (appending them to the vocabulary if they aren't already there). We recommend the following interface:

[3]See en.wikipedia.org/wiki/Specials_(Unicode_block)#Replacement_character for more information about the Unicode replacement character.

`def __init__(self, vocab, merges, special_tokens=None)` Construct a tokenizer from a given vocabulary, list of merges, and (optionally) a list of special tokens. This function should accept the following parameters:

`vocab: dict[int, bytes]`

`merges: list[tuple[bytes, bytes]]`

`special_tokens: list[str] | None = None`

`def from_files(cls, vocab_filepath, merges_filepath, special_tokens=None)` Class method that constructs and returns a `Tokenizer` from a serialized vocabulary and list of merges (in the same format that your BPE training code output) and (optionally) a list of special tokens. This method should accept the following additional parameters:

`vocab_filepath: str`

`merges_filepath: str`

`special_tokens: list[str] | None = None`

`def encode(self, text: str) -> list[int]` Encode an input text into a sequence of token IDs.

`def encode_iterable(self, iterable: Iterable[str]) -> Iterator[int]` Given an iterable of strings (e.g., a Python file handle), return a generator that lazily yields token IDs. This is required for memory-efficient tokenization of large files that we cannot directly load into memory.

`def decode(self, ids: list[int]) -> str` Decode a sequence of token IDs into text.

To test your `Tokenizer` against our provided tests, you will first need to implement the test adapter at [`adapters.get_tokenizer`]. Then, run `uv run pytest tests/test_tokenizer.py`. Your implementation should be able to pass all tests.

## 2.7 Experiments

**Problem (`tokenizer_experiments`): Experiments with tokenizers (4 points)**

---

(a) Sample 10 documents from TinyStories and OpenWebText. Using your previously-trained TinyStories and OpenWebText tokenizers (10K and 32K vocabulary size, respectively), encode these sampled documents into integer IDs. What is each tokenizer's compression ratio (bytes/token)?

**Deliverable**: A one-to-two sentence response.

(b) What happens if you tokenize your OpenWebText sample with the TinyStories tokenizer? Compare the compression ratio and/or qualitatively describe what happens.

**Deliverable**: A one-to-two sentence response.

(c) Estimate the throughput of your tokenizer (e.g., in bytes/second). How long would it take to tokenize the Pile dataset (825GB of text)?

**Deliverable**: A one-to-two sentence response.

(d) Using your TinyStories and OpenWebText tokenizers, encode the respective training and development datasets into a sequence of integer token IDs. We'll use this later to train our

language model. We recommend serializing the token IDs as a NumPy array of datatype `uint16`. Why is `uint16` an appropriate choice?

**Deliverable**: A one-to-two sentence response.

# 3 Transformer Language Model Architecture

A language model takes as input a batched sequence of integer token IDs (i.e., `torch.Tensor` of shape `(batch_size, sequence_length)`), and returns a (batched) normalized probability distribution over the vocabulary (i.e., a PyTorch Tensor of shape `(batch_size, sequence_length, vocab_size)`), where the predicted distribution is over the next word for each input token. When training the language model, we use these next-word predictions to calculate the cross-entropy loss between the actual next word and the predicted next word. When generating text from the language model during inference, we take the predicted next-word distribution from the final time step (i.e., the last item in the sequence) to generate the next token in the sequence (e.g., by taking the token with the highest probability, sampling from the distribution, etc.), add the generated token to the input sequence, and repeat.

In this part of the assignment, you will build this Transformer language model from scratch. We will begin with a high-level description of the model before progressively detailing the individual components.

## 3.1 Transformer LM

Given a sequence of token IDs, the Transformer language model uses an input embedding to convert token IDs to dense vectors, passes the embedded tokens through `num_layers` Transformer blocks, and then applies a learned linear projection (the "output embedding" or "LM head") to produce the predicted next-token logits. See Figure 1 for a schematic representation.

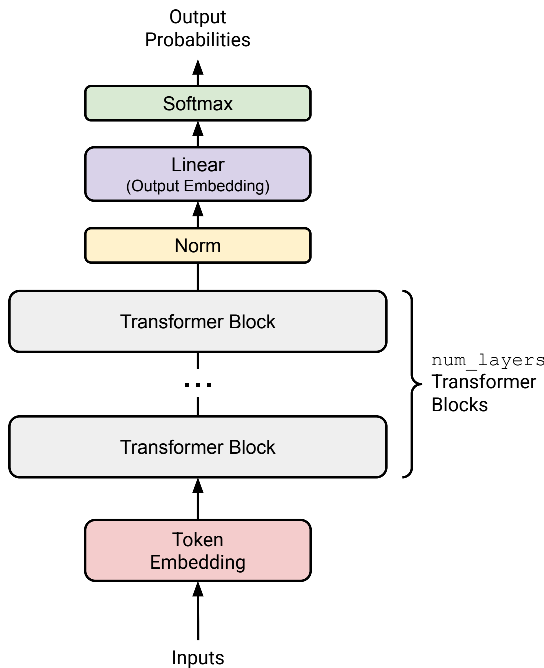


Figure 1: An overview of our Transformer language model.

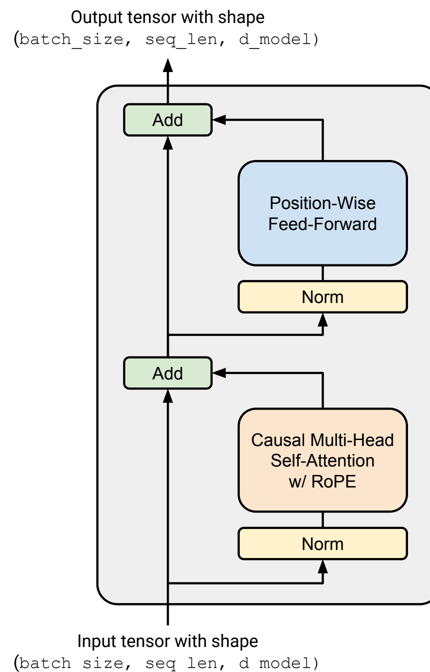


Figure 2: A pre-norm Transformer block.

#### Token Embeddings

In the very first step, the Transformer *embeds* the (batched) sequence of token IDs into a sequence of vectors containing information on the token identity (red blocks in Figure 1).

More specifically, given a sequence of token IDs, the Transformer language model uses a token embedding layer to produce a sequence of vectors. Each embedding layer takes in a tensor of integers of shape `(batch_size, sequence_length)` and produces a sequence of vectors of shape `(batch_size, sequence_length, d_model)`.

#### Pre-norm Transformer Block

After embedding, the activations are processed by several identically structured neural net layers. A standard decoder-only Transformer language model consists of `num_layers` identical layers (commonly called Transformer “blocks”). Each Transformer block takes in an input of shape `(batch_size, sequence_length, d_model)` and returns an output of shape `(batch_size, sequence_length, d_model)`. Each block aggregates information across the sequence (via self-attention) and non-linearly transforms it (via the feed-forward layers).

After `num_layers` Transformer blocks, we will take the final activations and turn them into a distribution over the vocabulary.

We will implement the “pre-norm” Transformer block (detailed in Section 3.4), which additionally requires the use of layer normalization (detailed below) after the final Transformer block to ensure its outputs are properly scaled.

After this normalization, we will use a standard learned linear transformation to convert the output of the Transformer blocks into predicted next-token logits (see, e.g., A. Radford et al. [7] equation 2).

## 3.2 Remark: Batching, Einsum and Efficient Computation

Throughout the Transformer, we will be performing the same computation applied to many batch-like inputs. Here are a few examples:

- **Elements of a batch**: we apply the same Transformer `forward` operation on each batch element.
- **Sequence length**: the “position-wise” operations like RMSNorm and feed-forward operate identically on each position of a sequence.
- **Attention heads**: the attention operation is batched across attention heads in a “multi-headed” attention operation.

It is useful to have an ergonomic way of performing such operations in a way that fully utilizes the GPU, and is easy to read and understand. Many PyTorch operations can take in excess “batch-like” dimensions at the start of a tensor and repeat/broadcast the operation across these dimensions efficiently.

For instance, say we are doing a position-wise, batched operation. We have a “data tensor” $D$ of shape `(batch_size, sequence_length, d_model)`, and we would like to do a batched vector-matrix multiply against a matrix $A$ of shape `(d_model, d_model)`. In this case, `D @ A` will do a batched matrix multiply, which is an efficient primitive in PyTorch, where the `(batch_size, sequence_length)` dimensions are batched over.

Because of this, it is helpful to assume that your functions may be given additional batch-like dimensions and to keep those dimensions at the start of the PyTorch shape. To organize tensors so they can be batched in this manner, they might need to be shaped using many steps of `view`, `reshape` and `transpose`. This can be a bit of a pain, and it often gets hard to read what the code is doing and what the shapes of your tensors are.

A more ergonomic option is to use *einsum notation* within `torch.einsum`, or rather use framework-agnostic libraries like `einops` or `einx`. The two key ops are `einsum`, which can do tensor contractions with arbitrary dimensions of input tensors, and `rearrange`, which can reorder, concatenate, and split arbitrary dimensions. It turns out almost all operations in machine learning are some combination of dimension juggling and tensor contraction with the occasional (usually pointwise) nonlinear function. This means that a lot of your code can be more readable and flexible when using einsum notation.

We **strongly** recommend learning and using einsum notation for the class. Students who have not been exposed to einsum notation before should use `einops` (docs here), and students who are already comfortable with `einops` should learn the more general `einx` (here).[4] Both packages are already installed in the environment we've supplied.

Here we give some examples of how einsum notation can be used. These are a supplement to the documentation for einops, which you should read first.

**Example (`einstein_example1`): Batched matrix multiplication with `einops.einsum`**

```python
import torch
from einops import rearrange, einsum

## Basic implementation
Y = D @ A.T
# Hard to tell the input and output shapes and what they mean.
# What shapes can D and A have, and do any of these have unexpected behavior?

## Einsum is self-documenting and robust
#                           D                 A     ->          Y
Y = einsum(D, A, "batch sequence d_in, d_out d_in -> batch sequence d_out")

## Or, a batched version where D can have any leading dimensions but A is constrained.
Y = einsum(D, A, "... d_in, d_out d_in -> ... d_out")
```

**Example (`einstein_example2`): Broadcasted operations with `einops.rearrange`**

We have a batch of images, and for each image we want to generate 10 dimmed versions based on some scaling factor:

```python
images = torch.randn(64, 128, 128, 3)  # (batch, height, width, channel)
dim_by = torch.linspace(start=0.0, end=1.0, steps=10)

## Reshape and multiply
dim_value = rearrange(dim_by,    "dim_value              -> 1 dim_value 1 1 1")
images_rearr = rearrange(images, "b height width channel -> b 1 height width channel")
dimmed_images = images_rearr * dim_value

## Or in one go:
dimmed_images = einsum(
    images, dim_by,
    "batch height width channel, dim_value -> batch dim_value height width channel"
)
```

[4]It's worth noting that while `einops` has a great amount of support, `einx` is not as battle-tested. You should feel free to fall back to using `einops` with some more plain PyTorch if you find any limitations or bugs in `einx`.

**Example (`einstein_example3`): Pixel mixing with `einops.rearrange`**

Suppose we have a batch of images represented as a tensor of shape `(batch, height, width, channel)`, and we want to perform a linear transformation across all pixels of the image, but this transformation should happen independently for each channel. Our linear transformation is represented as a matrix $B$ of shape `(height * width, height * width)`.

```
channels_last = torch.randn(64, 32, 32, 3)  # (batch, height, width, channel)
B = torch.randn(32*32, 32*32)

## Rearrange an image tensor for mixing across all pixels
channels_last_flat = channels_last.view(
    -1, channels_last.size(1) * channels_last.size(2), channels_last.size(3)
)
channels_first_flat = channels_last_flat.transpose(1, 2)
channels_first_flat_transformed = channels_first_flat @ B.T
channels_last_flat_transformed = channels_first_flat_transformed.transpose(1, 2)
channels_last_transformed = channels_last_flat_transformed.view(*channels_last.shape)
```

Instead, using `einops`:

```
height = width = 32
## Rearrange replaces clunky torch view + transpose
channels_first = rearrange(
    channels_last,
    "batch height width channel -> batch channel (height width)"
)
channels_first_transformed = einsum(
    channels_first, B,
    "batch channel pixel_in, pixel_out pixel_in -> batch channel pixel_out"
)
channels_last_transformed = rearrange(
    channels_first_transformed,
    "batch channel (height width) -> batch height width channel",
    height=height, width=width
)
```

Or, if you're feeling crazy: all in one go using `einx.dot` (`einx` equivalent of `einops.einsum`)

```
height = width = 32
channels_last_transformed = einx.dot(
    "batch row_in col_in channel, (row_out col_out) (row_in col_in)"
    "-> batch row_out col_out channel",
    channels_last, B,
    col_in=width, col_out=width
)
```

The first implementation here could be improved by placing comments before and after to indicate what the input and output shapes are, but this is clunky and susceptible to bugs. With einsum notation, documentation *is* implementation!

Einsum notation can handle arbitrary input batching dimensions, but also has the key benefit of being *self-documenting*. It's much clearer what the relevant shapes of your input and output tensors are in code

that uses einsum notation. For the remaining tensors, you can consider using Tensor type hints, for instance using the `jaxtyping` library (not specific to JAX).

We will talk more about the performance implications of using einsum notation in assignment 2, but for now know that they're almost always better than the alternative!

#### 3.2.1 Mathematical Notation and Memory Ordering

Many machine learning papers use row vectors in their notation, which result in representations that mesh well with the row-major memory ordering used by default in NumPy and PyTorch. With row vectors, a linear transformation looks like

$$y = xW^{\top}, \tag{1}$$

for row-major $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ and row-vector $x \in \mathbb{R}^{1 \times d_{\text{in}}}$. Notice that this lets us batch inputs by increasing the outermost dimension of $x$, meaning we can substitute vector input $x$ for matrix input $X \in \mathbb{R}^{\text{batch} \times d_{\text{in}}}$.

In linear algebra it's generally more common to use column vectors, where linear transformations look like

$$y = Wx, \tag{2}$$

given a row-major $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ and column-vector $x \in \mathbb{R}^{d_{\text{in}}}$. To batch the input in this setting, the batch dimension to $x$ would have to come last, so $x$ would need to be replaced with a matrix $\tilde{X} \in \mathbb{R}^{d_{\text{in}} \times \text{batch}}$.

**We will use mostly column vectors** for mathematical notation in this assignment, since math generally follows this notation. You should keep in mind that if you want to use plain matrix multiplication notation, you will have to apply matrices with a transpose as seen in the row vector convention in Equation 1, since PyTorch uses row-major memory ordering. If you use `einsum` for your linear algebra operations, this should be a non-issue as long as you label your axes correctly. As an aside, it's worth noting that other languages/linear algebra packages like Matlab, Julia and Fortran all use column-major memory ordering, meaning *batching dimensions come last*, but Python and related packages have adopted the C-standard of row-major ordering.

### 3.3 Basic Building Blocks: Linear and Embedding Modules

#### 3.3.1 Parameter Initialization

Training neural networks effectively often requires careful initialization of the model parameters—bad initializations can lead to undesirable behavior such as vanishing or exploding gradients. Pre-norm transformers are unusually robust to initializations, but they can still have a significant impact on training speed and convergence. Since this assignment is already long, we will save the details for assignment 3, and instead give you some approximate initializations that should work well for most cases. For now, use:

- Linear weights: $\mathcal{N}\left(\mu = 0, \sigma^2 = \frac{2}{d_{\text{in}} + d_{\text{out}}}\right)$ truncated at $[-3\sigma, 3\sigma]$.
- Embedding: $\mathcal{N}(\mu = 0, \sigma^2 = 1)$ truncated at $[-3, 3]$
- RMSNorm: $\mathbb{1}$

You should use `torch.nn.init.trunc_normal_` to initialize the truncated normal weights.

#### 3.3.2 Linear Module

Linear layers are a fundamental building block of Transformers and neural nets in general. First, you will implement your own `Linear` class that inherits from `torch.nn.Module` and performs a linear transformation:

$$y = Wx. \tag{3}$$

Note that we do not include a bias term, following most modern LLMs.

**Problem (`linear`): Implementing the linear module (1 point)**

---

**Deliverable**: Implement a `Linear` class that inherits from `torch.nn.Module` and performs a linear transformation. Your implementation should follow the interface of PyTorch's built-in `nn.Linear` module, except for not having a `bias` argument or parameter. We recommend the following interface:

**`def __init__(self, in_features, out_features, device=None, dtype=None)`** Construct a linear transformation module. This function should accept the following parameters:

**`in_features: int`** final dimension of the input

**`out_features: int`** final dimension of the output

**`device: torch.device | None = None`** Device to store the parameters on

**`dtype: torch.dtype | None = None`** Data type of the parameters

**`def forward(self, x: torch.Tensor) -> torch.Tensor`** Apply the linear transformation to the input.

Make sure to:

- subclass `nn.Module`
- call the superclass constructor
- construct and store your parameter as $W$ (not $W^\top$), putting it in an `nn.Parameter`
- of course, don't use `nn.Linear` or `nn.functional.linear`

For initializations, use the settings from above along with `torch.nn.init.trunc_normal_` to initialize the weights.

To test your `Linear` module, implement the test adapter at [`adapters.run_linear`]. The adapter should load the given weights into your `Linear` module. You can use `Module.load_state_dict` for this purpose. Then, run `uv run pytest -k test_linear`.

### 3.3.3 Embedding Module

As discussed above, the first layer of the Transformer is an embedding layer that maps integer token IDs into a vector space of dimension `d_model`. We will implement a custom `Embedding` class that inherits from `torch.nn.Module` (so you should not use `nn.Embedding`). The `forward` method should select the embedding vector for each token ID by indexing into an embedding matrix of shape `(vocab_size, d_model)` using a `torch.LongTensor` of token IDs with shape `(batch_size, sequence_length)`.

**Problem (`embedding`): Implement the embedding module (1 point)**

---

**Deliverable**: Implement the `Embedding` class that inherits from `torch.nn.Module` and performs an embedding lookup. Your implementation should follow the interface of PyTorch's built-in `nn.Embedding` module. We recommend the following interface:

`def __init__(self, num_embeddings, embedding_dim, device=None, dtype=None)` Construct an embedding module. This function should accept the following parameters:

`num_embeddings: int` Size of the vocabulary

`embedding_dim: int` Dimension of the embedding vectors, i.e., $d_{\text{model}}$

`device: torch.device | None = None` Device to store the parameters on

`dtype: torch.dtype | None = None` Data type of the parameters

`def forward(self, token_ids: torch.Tensor) -> torch.Tensor` Lookup the embedding vectors for the given token IDs.

Make sure to:

- subclass `nn.Module`
- call the superclass constructor
- initialize your embedding matrix as an `nn.Parameter`
- store the embedding matrix with the `d_model` being the final dimension
- of course, don't use `nn.Embedding` or `nn.functional.embedding`

Again, use the settings from above for initialization, and use `torch.nn.init.trunc_normal_` to initialize the weights.

To test your implementation, implement the test adapter at [`adapters.run_embedding`] . Then, run `uv run pytest -k test_embedding`.

## 3.4 Pre-Norm Transformer Block

Each Transformer block has two sub-layers: a multi-head self-attention mechanism and a position-wise feed-forward network ([A. Vaswani et al., 2017], section 3.1).

In the original Transformer paper, the model uses a residual connection around each of the two sub-layers, followed by layer normalization. This architecture is commonly known as the "post-norm" Transformer, since layer normalization is applied to the sub-layer output. However, a variety of work has found that moving layer normalization from the output of each sub-layer to the input of each sub-layer (with an additional layer normalization after the final Transformer block) improves Transformer training stability [T. Q. Nguyen et al., 2019; R. Xiong et al., 2020] — see Figure 2 for a visual representation of this "pre-norm" Transformer block. The output of each Transformer block sub-layer is then added to the sub-layer input via the residual connection (A. Vaswani et al. [8], section 5.4). An intuition for pre-norm is that there is a clean "residual stream" without any normalization going from the input embeddings to the final output of the Transformer, which is purported to improve gradient flow. This pre-norm Transformer is now the standard used in language models today (e.g., GPT-3, LLaMA, PaLM, etc.), so we will implement this variant. We will walk through each of the components of a pre-norm Transformer block, implementing them in sequence.

### 3.4.1 Root Mean Square Layer Normalization

The original Transformer implementation of A. Vaswani et al. [8] uses layer normalization [J. L. Ba et al., 2016] to normalize activations. Following H. Touvron et al. [12], we will use root mean square layer normalization (RMSNorm; B. Zhang et al. [13], equation 4) for layer normalization. Given a vector $a \in \mathbb{R}^{d_{\text{model}}}$ of activations, RMSNorm will rescale each activation $a_i$ as follows:

$$\text{RMSNorm}(a_i) = \frac{a_i}{\text{RMS}(a)} g_i, \tag{4}$$

where $\text{RMS}(a) = \sqrt{\frac{1}{d_\text{model}} \sum_{i=1}^{d_\text{model}} a_i^2 + \varepsilon}$. Here, $g_i$ is a learnable "gain" parameter (there are `d_model` such parameters total), and $\varepsilon$ is a hyperparameter that is often fixed at 1e-5.

You should upcast your input to `torch.float32` to prevent overflow when you square the input. Overall, your `forward` method should look like:

```
in_dtype = x.dtype
x = x.to(torch.float32)

# Your code here performing RMSNorm
...
result = ...

# Return the result in the original dtype
return result.to(in_dtype)
```

**Problem (`rmsnorm`): Root Mean Square Layer Normalization (1 point)**

---

**Deliverable**: Implement RMSNorm as a `torch.nn.Module`. We recommend the following interface:

**`def __init__(self, d_model: int, eps: float = 1e-5, device=None, dtype=None)`** Construct the RMSNorm module. This function should accept the following parameters:

**`d_model: int`** Hidden dimension of the model

**`eps: float = 1e-5`** Epsilon value for numerical stability

**`device: torch.device | None = None`** Device to store the parameters on

**`dtype: torch.dtype | None = None`** Data type of the parameters

**`def forward(self, x: torch.Tensor) -> torch.Tensor`** Process an input tensor of shape `(batch_size, sequence_length, d_model)` and return a tensor of the same shape.

**Note:** Remember to upcast your input to `torch.float32` before performing the normalization (and later downcast to the original dtype), as described above.

To test your implementation, implement the test adapter at [`adapters.run_rmsnorm`] . Then, run `uv run pytest -k test_rmsnorm`.

### 3.4.2 Position-Wise Feed-Forward Network

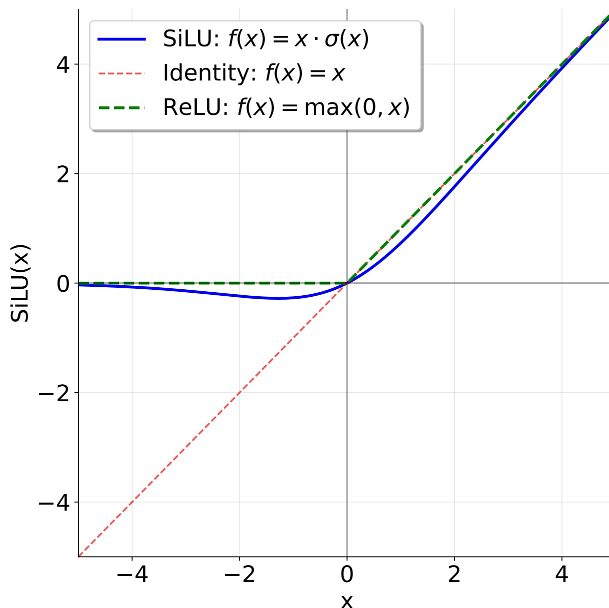


Figure 3: Comparing the SiLU (aka Swish) and ReLU activation functions.

In the original Transformer paper (section 3.3 of A. Vaswani et al. [8]), the Transformer feed-forward network consists of two linear transformations with a ReLU activation ($\mathrm{ReLU}(x) = \max(0, x)$) between them. In that original architecture, the dimensionality of the inner feed-forward layer is typically 4x the input dimensionality.

However, modern language models tend to incorporate two main changes compared to this original design: they use another activation function and employ a gating mechanism. Specifically, we will implement the "SwiGLU" activation function adopted in LLMs like Llama 3 [A. Grattafiori et al., 2024] and Qwen 2.5 [A. Yang et al., 2024], which combines the SiLU (often called Swish) activation with a gating mechanism called a Gated Linear Unit (GLU). We will also omit the bias terms sometimes used in linear layers, following most modern LLMs since PaLM [A. Chowdhery et al., 2022] and LLaMA [H. Touvron et al., 2023].

The SiLU or Swish activation function [D. Hendrycks et al., 2016; S. Elfwing et al., 2017] is defined as follows:

$$\mathrm{SiLU}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}} \tag{5}$$

As can be seen in Figure 3, the SiLU activation function is similar to the ReLU activation function, but is smooth at zero.

Gated Linear Units (GLUs) were originally defined by Y. N. Dauphin et al. [19] as the element-wise product of a linear transformation passed through a sigmoid function and another linear transformation:

$$\mathrm{GLU}(x, W_1, W_2) = \sigma(W_1 x) \odot W_2 x, \tag{6}$$

where $\odot$ represents element-wise multiplication. Gated Linear Units are suggested to "reduce the vanishing gradient problem for deep architectures by providing a linear path for the gradients while retaining non-linear capabilities."

Putting the SiLU/Swish and GLU together, we get the SwiGLU, which we will use for our feed-forward networks:

$$\text{FFN}(x) = \text{SwiGLU}(x, W_1, W_2, W_3) = W_2(\text{SiLU}(W_1 x) \odot W_3 x), \tag{7}$$

where $x \in \mathbb{R}^{d_{\text{model}}}$, $W_1, W_3 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$, $W_2 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$, and canonically, $d_{\text{ff}} = \frac{8}{3} d_{\text{model}}$. For concrete implementations, it is fine to round this to a nearby multiple of 64 for hardware efficiency.

N. Shazeer [20] first proposed combining the SiLU/Swish activation with GLUs and conducted experiments showing that SwiGLU outperforms baselines like ReLU and SiLU (without gating) on language modeling tasks. Later in the assignment, you will compare SwiGLU and SiLU. Though we've mentioned some heuristic arguments for these components (and the papers provide more supporting evidence), it's good to keep an empirical perspective: a now famous quote from Shazeer's paper is

> "We offer no explanation as to why these architectures seem to work; we attribute their success, as all else, to divine benevolence."

**Problem (`positionwise_feedforward`): Implement the position-wise feed-forward network (2 points)**

---

**Deliverable**: Implement the SwiGLU feed-forward network, composed of a SiLU activation function and a GLU.

**Note:** in this particular case, you should feel free to use `torch.sigmoid` in your implementation for numerical stability.

You should set $d_{\text{ff}}$ to approximately $\frac{8}{3} \times d_{\text{model}}$ in your implementation, while ensuring that the dimensionality of the inner feed-forward layer is a multiple of 64 to make good use of your hardware. To test your implementation against our provided tests, you will need to implement the test adapter at [`adapters.run_swiglu`]. Then, run `uv run pytest -k test_swiglu` to test your implementation.

### 3.4.3 Relative Positional Embeddings

To inject positional information into the model, we will implement Rotary Position Embeddings [J. Su et al., 2021], often called RoPE. For a given query token $q^{(i)} = W_q x^{(i)} \in \mathbb{R}^d$ at token position $i$, we will apply a pairwise rotation matrix $R^i$, giving us $q'^{(i)} = R^i q^{(i)} = R^i W_q x^{(i)}$. Here, $R^i$ will rotate pairs of embedding elements $q^{(i)}_{2k-1:2k}$ as 2d vectors by the angle $\theta_{i,k} = \frac{i}{\Theta^{(2k-2)/d}}$ for $k \in \{1, ..., d/2\}$ and some constant $\Theta$. Thus, we can consider $R^i$ to be a block-diagonal matrix of size $d \times d$, with blocks $R^i_k$ for $k \in \{1, ..., \frac{d}{2}\}$, with

$$R^i_k = \begin{pmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{pmatrix} \tag{8}$$

Thus we get the full rotation matrix

$$R^i = \begin{pmatrix} R_1^i & 0 & 0 & \cdots & 0 \\ 0 & R_2^i & 0 & \cdots & 0 \\ 0 & 0 & R_3^i & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & R_{d/2}^i \end{pmatrix}, \tag{9}$$

where 0s represent $2 \times 2$ zero matrices. While one could construct the full $d \times d$ matrix, a good solution should use the properties of this matrix to implement the transformation more efficiently. Since we only care about the relative rotation of tokens within a given sequence, we can reuse the values we compute for $\cos(\theta_{i,k})$ and $\sin(\theta_{i,k})$ across layers, and different batches. If you would like to optimize it, you may use a single RoPE module referenced by all layers, and it can have a 2d pre-computed buffer of sin and cos values created during init with `self.register_buffer(persistent=False)`, instead of an `nn.Parameter` (because we do not want to learn these fixed cosine and sine values). The exact same rotation process we did for our $q^{(i)}$ is then done for $k^{(j)}$, rotating by the corresponding $R^j$. Notice that this layer has no learnable parameters.

**Problem (`rope`): Implement RoPE (2 points)**

---

**Deliverable**: Implement a class `RotaryPositionalEmbedding` that applies RoPE to the input tensor.

The following interface is recommended:

`def __init__(self, theta: float, d_k: int, max_seq_len: int, device=None)` Construct the RoPE module and create buffers if needed.

`theta: float` $\Theta$ value for the RoPE

`d_k: int` dimension of query and key vectors

`max_seq_len: int` Maximum sequence length that will be input

`device: torch.device | None = None` Device to store the buffer on

`def forward(self, x: torch.Tensor, token_positions: torch.Tensor) -> torch.Tensor` Process an input tensor of shape `(..., seq_len, d_k)` and return a tensor of the same shape. Note that you should tolerate $x$ with an arbitrary number of batch dimensions. You should assume that the token positions are a tensor of shape `(..., seq_len)` specifying the token positions of $x$ along the sequence dimension.

You should use the token positions to slice your (possibly precomputed) cos and sin tensors along the sequence dimension.

To test your implementation, complete [`adapters.run_rope`] and make sure it passes `uv run pytest -k test_rope`.

### 3.4.4 Scaled Dot-Product Attention

We will now implement scaled dot-product attention as described in A. Vaswani et al. [8] (section 3.2.1). As a preliminary step, the definition of the Attention operation will make use of softmax, an operation that takes an unnormalized vector of scores and turns it into a normalized distribution:

$$\text{softmax}(v)_i = \frac{\exp(v_i)}{\sum_{j=1}^{n} \exp(v_j)}. \tag{10}$$

Note that $\exp(v_i)$ can become `inf` for large values (then, $\frac{\text{inf}}{\text{inf}} = \text{NaN}$). We can avoid this by noticing that the softmax operation is invariant to adding any constant $c$ to all inputs. We can leverage this property for numerical stability—typically, we will subtract the largest entry of $v$ from all elements of $v$, making the new largest entry 0. You will now implement softmax, using this trick for numerical stability.

**Problem (`softmax`): Implement softmax (1 point)**

---

**Deliverable**: Write a function to apply the softmax operation on a tensor. Your function should take two parameters: a tensor and a *dimension* $i$, and apply softmax to the $i$-th dimension of the input tensor. The output tensor should have the same shape as the input tensor, but its $i$-th dimension will now have a normalized probability distribution. Use the trick of subtracting the maximum value in the $i$-th dimension from all elements of the $i$-th dimension to avoid numerical stability issues.

To test your implementation, complete [`adapters.run_softmax`] and make sure it passes `uv run pytest -k test_softmax_matches_pytorch`.

We can now define the Attention operation mathematically as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{11}$$

where $Q \in \mathbb{R}^{n \times d_k}$, $K \in \mathbb{R}^{m \times d_k}$, and $V \in \mathbb{R}^{m \times d_v}$. Here, $Q$, $K$ and $V$ are all inputs to this operation—note that these are not the learnable parameters.

**Masking:** It is sometimes convenient to *mask* the output of an attention operation. A mask should have the shape $M \in \{\text{True}, \text{False}\}^{n \times m}$, and each row $i$ of this boolean matrix indicates which keys the query $i$ should attend to. Canonically (and slightly confusingly), a value of `True` at position $(i, j)$ indicates that the query $i$ *does* attend to the key $j$, and a value of `False` indicates that the query *does not* attend to the key. In other words, "information flows" at $(i, j)$ pairs with value `True`. For example, consider a $1 \times 3$ mask matrix with entries [[`True`, `True`, `False`]]. The single query vector attends only to the first two keys.

Computationally, it will be much more efficient to use masking than to compute attention on subsequences, and we can do this by taking the pre-softmax values $\left(QK^\top / \sqrt{d_k}\right)$ and adding a $-\infty$ to any entry of the mask matrix that is False.

**Problem (`scaled_dot_product_attention`): Implement scaled dot-product attention (5 points)**

---

**Deliverable**: Implement the scaled dot-product attention function. Your implementation should handle keys and queries of shape `(batch_size, ..., seq_len, d_k)` and values of shape `(batch_size, ..., seq_len, d_v)`, where `...` represents any number of other batch-like dimensions (if provided). The implementation should return an output with the shape `(batch_size, ..., seq_len, d_v)`. See Section 3.2 for a discussion on batch-like dimensions.

Your implementation should also support an optional user-provided boolean mask of shape `(seq_len, seq_len)`. The attention probabilities of positions with a mask value of `True` should collectively sum to 1, and the attention probabilities of positions with a mask value of `False` should be zero.

To test your implementation against our provided tests, you will need to implement the test adapter at [adapters.run_scaled_dot_product_attention]. `uv run pytest -k test_scaled_dot_product_attention` tests your implementation on third-order input tensors, while `uv run pytest -k test_4d_scaled_dot_product_attention` tests your implementation on fourth-order input tensors.

#### 3.4.5 Causal Multi-Head Self-Attention

We will implement multi-head self-attention as described in section 3.2.2 of A. Vaswani et al. [8]. Recall that, mathematically, the operation of applying multi-head attention is defined as follows:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h) \tag{12}$$

$$\text{for } \text{head}_i = \text{Attention}(Q_i, K_i, V_i) \tag{13}$$

with $Q_i$, $K_i$, $V_i$ being slice number $i \in \{1, ..., h\}$ of size $d_k$ or $d_v$ of the embedding dimension for $Q, K$, and $V$ respectively. With Attention being the scaled dot-product attention operation defined in Section 3.4.4. From this we can form the multi-head *self*-attention operation:

$$\text{MultiHeadSelfAttention}(x) = W_O \text{MultiHead}(W_Q x, W_K x, W_V x) \tag{14}$$

Here, the learnable parameters are $W_Q \in \mathbb{R}^{hd_k \times d_{\text{model}}}$, $W_K \in \mathbb{R}^{hd_k \times d_{\text{model}}}$, $W_V \in \mathbb{R}^{hd_v \times d_{\text{model}}}$, and $W_O \in \mathbb{R}^{d_{\text{model}} \times hd_v}$. Since the $Q$s, $K$s, and $V$s are sliced in the multi-head attention operation, we can think of $W_Q$, $W_K$ and $W_V$ as being separated for each head along the output dimension. When you have this working, you should be computing the key, value, and query projections in a total of three matrix multiplies.[5]

**Causal masking**

Your implementation should prevent the model from attending to future tokens in the sequence. In other words, if the model is given a token sequence $t_1, ..., t_n$, and we want to calculate the next-word predictions for the prefix $t_1, ..., t_i$ (where $i < n$), the model should *not* be able to access (attend to) the token representations at positions $t_{i+1}, ..., t_n$ since it will not have access to these tokens when generating text during inference (and these future tokens leak information about the identity of the true next word, trivializing the language modeling pre-training objective). For an input token sequence $t_1, ..., t_n$ we can naively prevent access to future tokens by running multi-head self-attention $n$ times (for the $n$ unique prefixes in the sequence). Instead, we'll use causal attention masking, which allows token $i$ to attend to all positions $j \leq i$ in the sequence. You can use `torch.triu` or a broadcasted index comparison to construct this mask, and you should take advantage of the fact that your scaled dot-product attention implementation from Section 3.4.4 already supports attention masking.

**Applying RoPE**

RoPE should be applied to the query and key vectors, but not the value vectors. Also, the head dimension should be handled as a batch dimension, because in multi-head attention, attention is being applied independently for each head. This means that precisely the same RoPE rotation should be applied to the query and key vectors for each head.

**Problem (`multihead_self_attention`): Implement causal multi-head self-attention (5 points)**

[5] As a stretch goal, try combining the key, query, and value projections into a single weight matrix so you only need a single matrix multiply.

**Deliverable**: Implement causal multi-head self-attention as a `torch.nn.Module`. Your implementation should accept (at least) the following parameters:

**`d_model:`** `int` Dimensionality of the Transformer block inputs.

**`num_heads:`** `int` Number of heads to use in multi-head self-attention.

Following A. Vaswani et al. [8], set $d_k = d_v = \frac{d_{\text{model}}}{h}$. To test your implementation against our provided tests, implement the test adapter at [`adapters.run_multihead_self_attention`] . Then, run `uv run pytest -k test_multihead_self_attention` to test your implementation.

## 3.5 The Full Transformer LM

Let's begin by assembling the Transformer block (it will be helpful to refer back to Figure 2). A Transformer block contains two 'sub-layers', one for the multihead self attention, and another for the SwiGLU feed-forward network. In each sub-layer, we first perform RMSNorm, then the main operation (MHA/FF), finally adding in the residual connection.

To be concrete, the first half (the first 'sub-layer') of the Transformer block should be implementing the following set of updates to produce an output $y$ from an input $x$,

$$y = x + \text{MultiHeadSelfAttention}(\text{RMSNorm}(x)). \tag{15}$$

**Problem (`transformer_block`): Implement the Transformer block (3 points)**

---

Implement the pre-norm Transformer block as described in Section 3.4 and illustrated in Figure 2. Your Transformer block should accept (at least) the following parameters.

**`d_model:`** `int` Dimensionality of the Transformer block inputs.

**`num_heads:`** `int` Number of heads to use in multi-head self-attention.

**`d_ff:`** `int` Dimensionality of the position-wise feed-forward inner layer.

To test your implementation, implement the adapter [`adapters.run_transformer_block`] . Then run `uv run pytest -k test_transformer_block` to test your implementation.

**Deliverable**: Transformer block code that passes the provided tests.

Now we put the blocks together, following the high-level diagram in Figure 1. Follow our description of the embedding in Section 3.1.0.1, feed this into `num_layers` Transformer blocks, and then pass that into the final layer norm and LM head to obtain an unnormalized distribution over the vocabulary (the logits).

**Problem (`transformer_lm`): Implementing the Transformer LM (3 points)**

---

Time to put it all together! Implement the Transformer language model as described in Section 3.1 and illustrated in Figure 1. At minimum, your implementation should accept all the aforementioned construction parameters for the Transformer block, as well as these additional parameters:

**`vocab_size:`** `int` The size of the vocabulary, necessary for determining the dimensionality of the token embedding matrix.

**context_length: int** The maximum context length, necessary for determining the dimensionality of the RoPE sin and cos buffer.

**num_layers: int** The number of Transformer blocks to use.

To test your implementation against our provided tests, you will first need to implement the test adapter at [adapters.run_transformer_lm] . Then, run `uv run pytest -k test_transformer_lm` to test your implementation.

**Deliverable**: A Transformer LM module that passes the above tests.

**Resource accounting**

It is useful to be able to understand how the various parts of the Transformer consume compute and memory. We will go through the steps to do some basic "FLOPs accounting." The *vast* majority of FLOPS in a Transformer are matrix multiplies, so our core approach is simple:

1. Write down all the matrix multiplies in a Transformer forward pass.
2. Convert each matrix multiply into FLOPs required.

For this second step, the following fact will be useful:

**Rule:** Given $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$, the matrix-matrix product $AB$ requires $2mnp$ FLOPs.

To see this, note that $(AB)[i, j] = A[i, :] \cdot B[:, j]$, and that this dot product requires $n$ additions and $n$ multiplications ($2n$ FLOPs). Then, since the matrix-matrix product $AB$ has $m \times p$ entries, the total number of FLOPS is $(2n)(mp) = 2mnp$.

Now, before you do the next problem, it can be helpful to go through each component of your Transformer block and Transformer LM, and list out all the matrix multiplies and their associated FLOPs costs.

**Problem (`transformer_accounting`): Transformer LM resource accounting (5 points)**

(a) Consider a GPT-2 XL-sized model using our assignment architecture, which has the following configuration:

**vocab_size:** 50,257

**context_length:** 1,024

**num_layers:** 48

**d_model:** 1,600

**num_heads:** 25

**d_ff:** 4,288 (the nearest multiple of 64 to $\frac{8}{3} \times 1,600$)

Suppose we constructed our model using this configuration. How many trainable parameters would our model have? Assuming each parameter is represented using single-precision floating point, how much memory is required to just load this model?

**Deliverable**: A one-to-two sentence response.

(b) Identify the matrix multiplies required to complete a forward pass of our GPT-2 XL-shaped model. How many FLOPs do these matrix multiplies require in total? Assume that our input sequence has `context_length` tokens.

**Deliverable**: A list of matrix multiplies (with descriptions), and the total number of FLOPs required.

(c) Based on your analysis above, which parts of the model require the most FLOPs?

**Deliverable**: A one-to-two sentence response.

(d) Repeat your analysis with GPT-2 small (12 layers, 768 `d_model`, 12 heads), GPT-2 medium (24 layers, 1024 `d_model`, 16 heads), and GPT-2 large (36 layers, 1280 `d_model`, 20 heads). As the model size increases, which parts of the Transformer LM take up proportionally more or less of the total FLOPs?

**Deliverable**: For each model, provide a breakdown of model components and its associated FLOPs (as a proportion of the total FLOPs required for a forward pass). In addition, provide a one-to-two sentence description of how varying the model size changes the proportional FLOPs of each component.

(e) Take GPT-2 XL and increase the context length to 16,384. How does the total FLOPs for one forward pass change? How does the relative contribution of FLOPs of the model components change?

**Deliverable**: A one-to-two sentence response.

# 4 Training a Transformer LM

We now have the steps to preprocess the data (via tokenizer) and the model (Transformer). What remains is to build all of the code to support training. This consists of the following:

- **Loss:** we need to define the loss function (cross-entropy).
- **Optimizer:** we need to define the optimizer to minimize this loss (AdamW).
- **Training loop:** we need all the supporting infrastructure that loads data, saves checkpoints, and manages training.

## 4.1 Cross-entropy loss

Recall that the Transformer language model defines a distribution $p_\theta(x_{i+1} \mid x_{1:i})$ for each sequence $x$ of length $m+1$ and $i = 1, \ldots, m$. Given a training set $D$ consisting of sequences of length $m+1$, we define the standard cross-entropy (negative log-likelihood) loss function:

$$\ell(\theta; D) = \frac{1}{|D|\ m} \sum_{x \in D} \sum_{i=1}^{m} -\log p_\theta(x_{i+1} \mid x_{1:i}). \tag{16}$$

(Note that a single forward pass in the Transformer yields $p_\theta(x_{i+1} \mid x_{1:i})$ for *all* $i = 1, \ldots, m$.)

In particular, the Transformer computes logits $o_i \in \mathbb{R}^{\text{vocab_size}}$ for each position $i$, which results in:[6]

$$p(x_{i+1} \mid x_{1:i}) = \text{softmax}(o_i)[x_{i+1}] = \frac{\exp(o_i[x_{i+1}])}{\sum_{a=1}^{\text{vocab_size}} \exp(o_i[a])}. \tag{17}$$

The cross-entropy loss is generally defined with respect to the vector of logits $o_i \in \mathbb{R}^{\text{vocab_size}}$ and target $x_{i+1}$.[7]

[6]Note that $o_i[k]$ refers to value at index $k$ of the vector $o_i$.

[7]This corresponds to the cross-entropy between the Dirac delta distribution over $x_{i+1}$ and the predicted $\text{softmax}(o_i)$ distribution.

Implementing the cross-entropy loss requires some care with numerical issues, just like in the case of softmax.

**Problem (`cross_entropy`): Implement cross-entropy (1 point)**

---

**Deliverable**: Write a function to compute the cross-entropy loss, which takes in predicted logits ($o_i$) and targets ($x_{i+1}$) and computes the cross-entropy $\ell_i = -\log \mathrm{softmax}(o_i)[x_{i+1}]$. Your function should handle the following:

- Subtract the largest element for numerical stability.
- Cancel out log and exp whenever possible.
- Handle any additional batch dimensions and return the *average* across the batch. As with Section 3.2, we assume batch-like dimensions always come first, before the vocabulary size dimension.

Implement [`adapters.run_cross_entropy`] , then run `uv run pytest -k test_cross_entropy` to test your implementation.

**Perplexity**

Cross-entropy suffices for training, but when we evaluate the model, we also want to report perplexity. For a sequence of length $m$ where we suffer cross-entropy losses $\ell_1, ..., \ell_m$:

$$\text{perplexity} = \exp\left(\frac{1}{m}\sum_{i=1}^{m}\ell_i\right). \tag{18}$$

## 4.2 The SGD Optimizer

Now that we have a loss function, we will begin our exploration of optimizers. The simplest gradient-based optimizer is Stochastic Gradient Descent (SGD). We start with randomly initialized parameters $\theta_0$. Then for each step $t = 0, ..., T-1$, we perform the following update:

$$\theta_{t+1} \leftarrow \theta_t - \alpha_t \nabla L(\theta_t; B_t), \tag{19}$$

where $B_t$ is a random batch of data sampled from the dataset $D$, and the *learning rate* $\alpha_t$ and *batch size* $|B_t|$ are hyperparameters.

### 4.2.1 Implementing SGD in PyTorch

To implement our optimizers, we will subclass the PyTorch `torch.optim.Optimizer` class. An `Optimizer` subclass must implement two methods:

**`def __init__(self, params, ...)`** should initialize your optimizer. Here, `params` will be a collection of parameters to be optimized (or parameter groups, in case the user wants to use different hyperparameters, such as learning rates, for different parts of the model). Make sure to pass `params` to the `__init__` method of the base class, which will store these parameters for use in `step`. You can take additional arguments depending on the optimizer (e.g., the learning rate is a common one), and pass them to the base class constructor as a dictionary, where keys are the names (strings) you choose for these parameters.

**`def step(self)`** should make one update of the parameters. During the training loop, this will be called after the backward pass, so you have access to the gradients on the last batch. This method should iterate through each parameter tensor `p` and modify them *in place*, i.e. setting `p.data`, which holds the

tensor associated with that parameter based on the gradient `p.grad` (if it exists), the tensor representing the gradient of the loss with respect to that parameter.

The PyTorch optimizer API has a few subtleties, so it's easier to explain it with an example. To make our example richer, we'll implement a slight variation of SGD where the learning rate decays over training, starting with an initial learning rate $\alpha$ and taking successively smaller steps over time:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{t+1}} \nabla L(\theta_t; B_t) \tag{20}$$

Let's see how this version of SGD would be implemented as a PyTorch `Optimizer`:

```
from collections.abc import Callable, Iterable
from typing import Optional
import torch
import math

class SGD(torch.optim.Optimizer):
    def __init__(self, params, lr=1e-3):
        if lr < 0:
            raise ValueError(f"Invalid learning rate: {lr}")
        defaults = {"lr": lr}
        super().__init__(params, defaults)

    def step(self, closure: Optional[Callable] = None):
        loss = None if closure is None else closure()
        for group in self.param_groups:
            lr = group["lr"]  # Get the learning rate.
            for p in group["params"]:
                if p.grad is None:
                    continue

                state = self.state[p]  # Get state associated with p.
                t = state.get("t", 0)  # Get iteration number from the state, or 0.
                grad = p.grad.data  # Get the gradient of loss with respect to p.
                p.data -= lr / math.sqrt(t + 1) * grad  # Update weight tensor in-place.
                state["t"] = t + 1  # Increment iteration number.

        return loss
```

In `__init__`, we pass the parameters, as well as default hyperparameters, to the base class constructor (the parameters might come in groups, each with different hyperparameters). In case the parameters are just a single collection of `torch.nn.Parameter` objects, the base constructor will create a single group and assign it the default hyperparameters. Then, in `step`, we iterate over each parameter group, then over each parameter in that group, and apply Equation 20. Here, we keep the iteration number as a state associated with each parameter: we first read this value, use it in the gradient update, and then update it. The API specifies that the user might pass in a callable `closure` to re-compute the loss before the optimizer step. We won't need this for the optimizers we'll use, but we add it to comply with the API.

To see this working, we can use the following minimal example of a *training loop*:

```
weights = torch.nn.Parameter(5 * torch.randn((10, 10)))
opt = SGD([weights], lr=1)

for t in range(100):
    opt.zero_grad()  # Reset the gradients for all learnable parameters.
    loss = (weights**2).mean() # Compute a scalar loss value.
    print(loss.cpu().item())
```

```
loss.backward() # Run backward pass, which computes gradients.
opt.step() # Run optimizer step.
```

This is the typical structure of a training loop: in each iteration, we will compute the loss and run a step of the optimizer. When training language models, our learnable parameters will come from the model (in PyTorch, `m.parameters()` gives us this collection). The loss will be computed over a sampled batch of data, but the basic structure of the training loop will be the same.

**Problem (`learning_rate_tuning`): Tuning the learning rate (1 point)**

As we will see, one of the hyperparameters that affects training the most is the learning rate. Let's see that in practice in our toy example. Run the SGD example above with three other values for the learning rate: 1e1, 1e2, and 1e3, for just 10 training iterations. What happens with the loss for each of these learning rates? Does it decay faster, slower, or does it diverge (i.e., increase over the course of training)?

**Deliverable**: A one-to-two sentence response with the behaviors you observed.

## 4.3 AdamW

Modern language models are typically trained with more sophisticated optimizers, instead of SGD. Most optimizers used recently are derivatives of the Adam optimizer [D. P. Kingma et al., 2015]. We will use AdamW [I. Loshchilov et al., 2019], which is in wide use in recent work. AdamW proposes a modification to Adam that improves regularization by adding *weight decay* (at each iteration, we pull the parameters towards 0), in a way that is decoupled from the gradient update. We will implement AdamW as described in algorithm 2 of I. Loshchilov et al. [23].

AdamW is *stateful*: for each parameter, it keeps track of a running estimate of its first and second moments. Thus, AdamW uses additional memory in exchange for improved stability and convergence. Besides the learning rate $\alpha$, AdamW has a pair of hyperparameters $(\beta_1, \beta_2)$ that control the updates to the moment estimates, and a weight decay rate $\lambda$. Typical applications set $(\beta_1, \beta_2)$ to $(0.9, 0.999)$, but large language models like LLaMA [H. Touvron et al., 2023] and GPT-3 [T. B. Brown et al., 2020] are often trained with $(0.9, 0.95)$. The algorithm can be written as follows, where $\varepsilon$ is a small value (e.g., $10^{-8}$) used to improve numerical stability in case we get extremely small values in $v$:

**Algorithm 1:** AdamW Optimizer

1 init($\theta$) ▷ Initialize learnable parameters
2 $m \leftarrow 0$ ▷ Initial value of the first moment vector; same shape as $\theta$
3 $v \leftarrow 0$ ▷ Initial value of the second moment vector; same shape as $\theta$
4 **for** $t = 1, ..., T$ **do**
5 Sample batch of data $B_t$
6 $g \leftarrow \nabla_\theta \ell(\theta; B_t)$ ▷ Compute the gradient of the loss
7 $\alpha_t \leftarrow \alpha \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t}$ ▷ Compute adjusted $\alpha$ for iteration $t$
8 $\theta \leftarrow \theta - \alpha\lambda\theta$ ▷ Apply weight decay
9 $m \leftarrow \beta_1 m + (1 - \beta_1) g$ ▷ Update the first moment estimate
10 $v \leftarrow \beta_2 v + (1 - \beta_2) g^2$ ▷ Update the second moment estimate
11 $\theta \leftarrow \theta - \alpha_t \frac{m}{\sqrt{v}+\varepsilon}$ ▷ Apply moment-adjusted weight updates
12 **end for**

Algorithm 1: AdamW optimizer pseudocode.

Note that $t$ starts at 1. You will now implement this optimizer.

**Problem (`adamw`): Implement AdamW (2 points)**

**Deliverable**: Implement the AdamW optimizer as a subclass of `torch.optim.Optimizer`. Your class should take the learning rate $\alpha$ in `__init__`, as well as the $\beta$, $\varepsilon$ and $\lambda$ hyperparameters. To help you keep state, the base `Optimizer` class gives you a dictionary `self.state`, which maps `nn.Parameter` objects to a dictionary that stores any information you need for that parameter (for AdamW, this would be the moment estimates). Implement [`adapters.get_adamw_cls`] and make sure it passes `uv run pytest -k test_adamw`.

**Problem (`adamw_accounting`): Resource accounting for training with AdamW (2 points)**

Let us compute how much memory and compute running AdamW requires. Assume we are using float32 for every tensor.

(a) How much peak memory does running AdamW require? Decompose your answer based on the memory usage of the parameters, activations, gradients, and optimizer state. Express your answer in terms of the `batch_size` and the model hyperparameters (`vocab_size`, `context_length`, `num_layers`, `d_model`, `num_heads`). Assume $d_{\text{ff}} = \frac{8}{3} \times d_{\text{model}}$.

For simplicity, when calculating memory usage of activations, consider only the following components:

- Transformer block
  - RMSNorm(s)
  - Multi-head self-attention sublayer: $QKV$ projections, $QK^\top$ matrix multiply, softmax, weighted sum of values, output projection.
  - Position-wise feed-forward (SwiGLU): $W_1$, $W_2$, SiLU on the gate branch, element-wise product, $W_3$

- final RMSNorm
- output embedding
- cross-entropy on logits

**Deliverable**: An algebraic expression for each of parameters, activations, gradients, and optimizer state, as well as the total.

(b) Instantiate your answer for a GPT-2 XL-shaped model to get an expression that only depends on the `batch_size`. What is the maximum batch size you can use and still fit within 80GB memory?

**Deliverable**: An expression that looks like $a \cdot \text{batch_size} + b$ for numerical values $a, b$, and a number representing the maximum batch size.

(c) How many FLOPs does running one step of AdamW take?

**Deliverable**: An algebraic expression, with a brief justification.

(d) Model FLOPs utilization (MFU) is defined as the ratio of observed throughput (tokens per second) relative to the hardware's theoretical peak FLOP throughput [A. Chowdhery et al., 2022]. An NVIDIA H100 GPU has a theoretical peak of 495 teraFLOP/s for "float32" (actually TensorFloat-32, which in reality is "bfloat19") operations. Assuming you are able to get 50% MFU, how long would it take to train a GPT-2 XL for 400K steps and a batch size of 1024 on a single H100? Following J. Kaplan et al. [25] and J. Hoffmann et al. [26], assume that the backward pass has twice the FLOPs of the forward pass.

**Deliverable**: The number of hours training would take, with a brief justification.

### 4.4 Learning rate scheduling

The value for the learning rate that leads to the quickest decrease in loss often varies during training. In training Transformers, it is typical to use a learning rate *schedule*, where we start with a bigger learning rate, making quicker updates in the beginning, and slowly decay it to a smaller value as the model trains.[8] In this assignment, we will implement the cosine annealing schedule used to train LLaMA [H. Touvron et al., 2023].

A scheduler is simply a function that takes the current step $t$ and other relevant parameters (such as the initial and final learning rates), and returns the learning rate to use for the gradient update at step $t$. The simplest schedule is the constant function, which will return the same learning rate given any $t$.

The cosine annealing learning rate schedule takes (i) the current iteration $t$, (ii) the maximum learning rate $\alpha_{\max}$, (iii) the minimum (final) learning rate $\alpha_{\min}$, (iv) the number of *warm-up* iterations $T_w$, and (v) the final iteration of cosine annealing $T_c$. The learning rate at iteration $t$ is defined as:

**(Warm-up)** If $t < T_w$, then $\alpha_t = \frac{t}{T_w}\alpha_{\max}$.

**(Cosine annealing)** If $T_w \le t \le T_c$, then $\alpha_t = \alpha_{\min} + \frac{1}{2}\left(1 + \cos\left(\frac{t-T_w}{T_c-T_w}\pi\right)\right)(\alpha_{\max} - \alpha_{\min})$.

**(Post-annealing)** If $t > T_c$, then $\alpha_t = \alpha_{\min}$.

**Problem (`learning_rate_schedule`): Implement cosine learning rate schedule with warmup (1 point)**

[8] It's sometimes common to use a schedule where the learning rate rises back up (restarts) to help get past local minima.

Write a function that takes $t$, $\alpha_{\max}$, $\alpha_{\min}$, $T_w$ and $T_c$, and returns the learning rate $\alpha_t$ according to the scheduler defined above. Then implement [adapters.get_lr_cosine_schedule] and make sure it passes `uv run pytest -k test_get_lr_cosine_schedule`.

### 4.5 Gradient clipping

During training, we can sometimes hit training examples that yield large gradients, which can destabilize training. To mitigate this, one technique often employed in practice is *gradient clipping*. The idea is to enforce a limit on the norm of the gradient after each backward pass before taking an optimizer step.

Given the gradient (for all parameters) $g$, we compute its $\ell_2$-norm $\|g\|_2$. If this norm is less than a maximum value $M$, then we leave $g$ as is; otherwise, we scale $g$ down by a factor of $\frac{M}{\|g\|_2+\varepsilon}$ (where a small $\varepsilon$, like $10^{-6}$, is added for numeric stability). Note that the resulting norm will be just under $M$.

**Problem (`gradient_clipping`): Implement gradient clipping (1 point)**

Write a function that implements gradient clipping. Your function should take a list of parameters and a maximum $\ell_2$-norm. It should modify each parameter gradient in place. Use $\varepsilon = 10^{-6}$ (the PyTorch default). Then, implement the adapter [adapters.run_gradient_clipping] and make sure it passes `uv run pytest -k test_gradient_clipping`.

## 5 Training loop

We will now finally put together the major components we've built so far: the tokenized data, the model, and the optimizer.

### 5.1 Data Loader

The tokenized data (e.g., that you prepared in `tokenizer_experiments`) is a single sequence of tokens $x = (x_1, ..., x_n)$. Even though the source data might consist of separate documents (e.g., different web pages, or source code files), a common practice is to concatenate all of those into a single sequence of tokens, adding a delimiter between them (such as the `<|endoftext|>` token).

A *data loader* turns this into a stream of *batches*, where each batch consists of $B$ sequences of length $m$, paired with the corresponding next tokens, also with length $m$. For example, for $B = 1, m = 3$, $([x_2, x_3, x_4], [x_3, x_4, x_5])$ would be one potential batch.

Loading data in this way simplifies training for a number of reasons. First, any $1 \leq i \leq n - m$ gives a valid training sequence, so sampling training sequences is trivial. Since all training sequences have the same length, there's no need to pad input sequences, which improves hardware utilization (also by increasing batch size $B$). Finally, we also don't need to load the full dataset to sample training data, making it easy to handle large datasets that might not otherwise fit in memory.

**Problem (`data_loading`): Implement data loading (2 points)**

**Deliverable**: Write a function that takes a numpy array $x$ (integer array with token IDs), a `batch_size`, a `context_length` and a PyTorch device string (e.g., `'cpu'` or `'cuda:0'`), and returns a pair of tensors: the sampled input sequences and the corresponding next-token targets. Both

tensors should have shape `(batch_size, context_length)` containing token IDs, and both should be placed on the requested device. To test your implementation against our provided tests, you will first need to implement the test adapter at [adapters.run_get_batch]. Then, run `uv run pytest -k test_get_batch` to test your implementation.

**Low-Resource Tip: Data loading on CPU or Apple Silicon**

---

If you are planning to train your LM on CPU or Apple Silicon, you need to move your data to the correct device (and similarly, you should use the same device for your model later on).

If you are on CPU, you can use the `'cpu'` device string, and on Apple Silicon (M* chips), you can use the `'mps'` device string.

For more on MPS, check out these resources:

- https://docs.pytorch.org/docs/stable/mps.html
- https://docs.pytorch.org/docs/stable/notes/mps.html
- https://developer.apple.com/documentation/metalperformanceshaders

What if the dataset is too big to load into memory? We can use a Unix system call named `mmap` which maps a file on disk to virtual memory, and lazily loads the file contents when that memory location is accessed. Thus, you can "pretend" you have the entire dataset in memory. Numpy implements this through `np.memmap` (or the flag `mmap_mode='r'` to `np.load`, if you originally saved the array with `np.save`), which will return a numpy array-like object that loads the entries on-demand as you access them. **When sampling from your dataset (i.e., a numpy array) during training, be sure to load the dataset in memory-mapped mode** (via `np.memmap` or the flag `mmap_mode='r'` to `np.load`, depending on how you saved the array). Make sure you also specify a `dtype` that matches the array that you're loading. It may be helpful to explicitly verify that the memory-mapped data looks correct (e.g., doesn't contain values beyond the expected vocabulary size).

## 5.2 Checkpointing

In addition to loading data, we will also need to save models as we train. When running jobs, we often want to be able to resume a training run that stopped midway through (e.g., due to your job timing out, machine failure, etc). Even when all goes well, we might also want to later have access to intermediate models (e.g., to study training dynamics post-hoc, take samples from models at different stages of training, etc).

A checkpoint should have all the states that we need to resume training. We of course want to be able to restore model weights at a minimum. If using a stateful optimizer (such as AdamW), we will also need to save the optimizer's state (e.g., in the case of AdamW, the moment estimates). Finally, to resume the learning rate schedule, we will need to know the iteration number we stopped at. PyTorch makes it easy to save all of these: every `nn.Module` has a `state_dict()` method that returns a dictionary with all learnable weights; we can restore these weights later with the sister method `load_state_dict()`. The same goes for any `torch.optim.Optimizer`. Finally, `torch.save(obj, dest)` can dump an object (e.g., a dictionary containing tensors as some values, but also regular Python objects like integers) to a file (path) or file-like object, which can then be loaded back into memory with `torch.load(src)`.

**Problem (`checkpointing`): Implement model checkpointing (1 point)**

Implement the following two functions to load and save checkpoints:

`def save_checkpoint(model, optimizer, iteration, out)` should dump all the state from the model, optimizer and iteration into the file-like object `out`. You can use the `state_dict` method of both the model and the optimizer to get their relevant states and use `torch.save(obj, out)` to dump `obj` into out (PyTorch supports either a path or a file-like object here). A typical choice is to have `obj` be a dictionary, but you can use whatever format you want as long as you can load your checkpoint later.

This function expects the following parameters:

```
model: torch.nn.Module
optimizer: torch.optim.Optimizer
iteration: int
out: str | os.PathLike | typing.BinaryIO | typing.IO[bytes]
```

`def load_checkpoint(src, model, optimizer)` should load a checkpoint from `src` (path or file-like object), and then recover the model and optimizer states from that checkpoint. Your function should return the iteration number that was saved to the checkpoint. You can use `torch.load(src)` to recover what you saved in your `save_checkpoint` implementation, and the `load_state_dict` method in both the model and optimizer to return them to their previous states.

This function expects the following parameters:

```
src: str | os.PathLike | typing.BinaryIO | typing.IO[bytes]
model: torch.nn.Module
optimizer: torch.optim.Optimizer
```

Implement the [`adapters.run_save_checkpoint`] and [`adapters.run_load_checkpoint`] adapters, and make sure they pass `uv run pytest -k test_checkpointing`.

## 5.3 Training loop

Now, it's finally time to put all of the components you implemented together into your main training script. It will pay off to make it easy to start training runs with different hyperparameters (e.g., by taking them as command-line arguments), since you will be doing these many times later to study how different choices impact training.

**Problem (`training_together`): Put it together (4 points)**

**Deliverable**: Write a script that runs a training loop to train your model on user-provided input. In particular, we recommend that your training script allow for (at least) the following:

- Ability to configure and control the various model and optimizer hyperparameters.
- Memory-efficient loading of large training and validation datasets with `np.memmap`.
- Serializing checkpoints to a user-provided path.

- Periodically logging training and validation performance (e.g., to console and/or an external service like Weights and Biases).[9]

# 6 Generating text

Now that we can train models, the last piece we need is the ability to generate text from our model. Recall that a language model takes in a (possibly batched) integer sequence of length `sequence_length` and produces a matrix of size `(sequence_length, vocab_size)`, where each element of the sequence is a probability distribution predicting the next token after that position. We will now write a few functions to turn this into a sampling scheme for new sequences.

### Softmax

By standard convention, the language model output is the output of the final linear layer (the "logits") and so we have to turn this into a normalized probability via the *softmax* operation, which we saw earlier in Equation 10.

### Decoding

To generate text (decode) from our model, we will provide the model with a sequence of prefix tokens (the "prompt"), and ask it to produce a probability distribution over the vocabulary that predicts the next token in the sequence. Then, we will sample from this distribution over the vocabulary items to determine the next output token.

Concretely, one step of the decoding process should take in a sequence $x_{1\ldots t}$ and return a token $x_{t+1}$ via the following equation,

$$P(x_{t+1} = i \mid x_{1\ldots t}) = \frac{\exp(v_i)}{\sum_j \exp(v_j)} \tag{21}$$

$$v = \text{TransformerLM}(x_{1\ldots t})_t \in \mathbb{R}^{\text{vocab_size}} \tag{22}$$

where TransformerLM is our model which takes as input a sequence of length `sequence_length` and produces a matrix of size `(sequence_length, vocab_size)`, and we take the last element of this matrix, as we are looking for the next token prediction at the $t$-th position.

This gives us a basic decoder by repeatedly sampling from these one-step conditionals (appending our previously-generated output token to the input of the next decoding timestep) until we generate the end-of-sequence token `<|endoftext|>` (or a user-specified maximum number of tokens to generate).

### Decoder tricks

We will be experimenting with small models, and small models can sometimes generate very low-quality texts. Two simple decoder tricks can help fix these issues. First, in *temperature scaling* we modify our softmax with a temperature parameter $\tau$, where the new softmax is

$$\text{softmax}(v, \tau)_i = \frac{\exp(v_i/\tau)}{\sum_{j=1}^{\text{vocab_size}} \exp(v_j/\tau)}. \tag{23}$$

Note how setting $\tau \to 0$ makes it so that the largest element of $v$ dominates, and the output of the softmax becomes a one-hot vector concentrated at this maximal element.

[9] wandb.ai

Second, another trick is *nucleus* or *top-p* sampling, where we modify the sampling distribution by truncating low-probability tokens. Let $q$ be a probability distribution that we get from a (temperature-scaled) softmax of size `vocab_size`. Nucleus sampling with hyperparameter $p$ produces the next token according to the equation

$$P(x_{t+1} = i \mid q) = \begin{cases} \frac{q_i}{\sum_{j \in V(p)} q_j} & \text{if } i \in V(p) \\ 0 & \text{otherwise} \end{cases} \tag{24}$$

where $V(p)$ is the *smallest* set of indices such that $\sum_{j \in V(p)} q_j \geq p$. You can compute this quantity easily by first sorting the probability distribution $q$ by magnitude, and selecting the largest vocabulary elements until you reach the target level of $p$.

**Problem (`decoding`): Decoding (3 points)**

---

**Deliverable**: Implement a function to decode from your language model. We recommend that you support the following features:

- Generate completions for a user-provided prompt (i.e., take in some $x_{1\ldots t}$ and sample a completion until you hit an `<|endoftext|>` token).
- Allow the user to control the maximum number of generated tokens.
- Given a desired temperature value, apply softmax temperature scaling to the predicted next-token distributions before sampling.
- Top-$p$ sampling ([A. Holtzman et al., 2020] also referred to as nucleus sampling), given a user-specified threshold value.

# 7 Experiments

Now it is time to put everything together and train (small) language models on a pretraining dataset.

## 7.1 How to Run Experiments and Deliverables

The best way to understand the rationale behind the architectural components of a Transformer is to actually modify it and run it yourself. There is no substitute for hands-on experience.

To this end, it's important to be able to experiment **quickly, consistently, and keep records** of what you did. To experiment quickly, we will be running many experiments on a small-scale model (about 17M total parameters) and simple dataset (TinyStories). To do things consistently, you will ablate components and vary hyperparameters in a systematic way, and to keep records we will ask you to submit a log of your experiments and learning curves associated with each experiment.

To make it possible to submit loss curves, **make sure to periodically evaluate validation losses and record both the number of steps and wall-clock times**. You might find logging infrastructure such as Weights and Biases helpful.

**Problem (`experiment_log`): Experiment logging (3 points)**

---

For your training and evaluation code, create experiment tracking infrastructure that allows you to track your experiments and loss curves with respect to gradient steps and wall-clock time.

**Deliverable**: Logging infrastructure code for your experiments and an experiment log (a document of all the things you tried) for the assignment problems below in this section.

## 7.2 TinyStories

We are going to start with a very simple dataset (TinyStories; R. Eldan et al. [1]) where models will train quickly, and we can see some interesting behaviors. The instructions for getting this dataset are in Section 1. An example of what this dataset looks like is below.

**Example (`tinystories_example`): One example from TinyStories**

Once upon a time there was a little boy named Ben. Ben loved to explore the world around him. He saw many amazing things, like beautiful vases that were on display in a store. One day, Ben was walking through the store when he came across a very special vase. When Ben saw it he was amazed! He said, "Wow, that is a really amazing vase! Can I buy it?" The shopkeeper smiled and said, "Of course you can. You can take it home and show all your friends how amazing it is!" So Ben took the vase home and he was so proud of it! He called his friends over and showed them the amazing vase. All his friends thought the vase was beautiful and couldn't believe how lucky Ben was. And that's how Ben found an amazing vase in the store!

### 7.2.1 Hyperparameter tuning

We will tell you some very basic hyperparameters to start with and ask you to find some settings for others that work well.

**Vocab size** 10000. Typical vocabulary sizes are in the tens to hundreds of thousands. You should vary this and see how the vocabulary and model behavior change.

**Context length** 256. Simple datasets such as TinyStories might not need long sequence lengths, but for the later OpenWebText data, you may want to vary this. Try varying this and seeing the impact on both the per-iteration runtime and the final perplexity.

**`d_model`** 512. This is slightly smaller than the 768 dimensions used in many small Transformer papers, but this will make things faster.

**`d_ff`** 1344. This is roughly $\frac{8}{3}d_{\text{model}}$ while being a multiple of 64, which is good for GPU performance.

**RoPE theta parameter $\Theta$** 10000.

**Number of layers and heads** 4 layers, 16 heads. Together, this will give about 17M non-embedding parameters which is a fairly small Transformer.

**Total tokens processed** 327,680,000 (your batch size × total step count × context length should equal roughly this value).

You should do some trial and error to find good defaults for the following other hyperparameters: `learning rate`, `learning rate warmup`, `other AdamW hyperparameters` $(\beta_1, \beta_2, \varepsilon)$, and `weight decay`. You can find some typical choices of such hyperparameters in D. P. Kingma et al. [22].

### 7.2.2 Putting it together

Now you can put everything together by getting a trained BPE tokenizer, tokenizing the training dataset, and running this in the training loop that you wrote. **Important note:** If your implementation is correct and efficient, the above hyperparameters should result in a roughly 20–30 minute runtime on 1 B200

GPU. If you have runtimes that are much longer, please check and make sure your dataloading, checkpointing, or validation loss code is not bottlenecking your runtimes and that your implementation is properly batched.

#### 7.2.3 Tips and tricks for debugging model architectures

We highly recommend getting comfortable with your IDE's built-in debugger (e.g., VSCode/Zed), which will save you time compared to debugging with print statements. If you use a text editor, you can use something like `ipdb`. A few other good practices when debugging model architectures are:

- A common first step when developing any neural net architecture is to overfit to a single minibatch. If your implementation is correct, you should be able to quickly drive the training loss to near-zero.
- Set debug breakpoints in various model components, and inspect the shapes of intermediate tensors to make sure they match your expectations.
- Monitor the norms of activations, model weights, and gradients to make sure they are not exploding or vanishing.

**Problem (`learning_rate`): Tune the learning rate (2 B200 hrs) (3 points)**

The learning rate is one of the most important hyperparameters to tune. Taking the base model you've trained, answer the following questions:

(a) Perform a hyperparameter sweep over the learning rates and report the final losses (or note divergence if the optimizer diverges).

**Deliverable**: Learning curves associated with multiple learning rates. Explain your hyperparameter search strategy.

**Deliverable**: A model with validation loss (per-token) on TinyStories of at most 1.45

**Low-Resource Tip: Train for a few steps on CPU or Apple Silicon**

If you are running on `cpu` or `mps`, you should instead reduce the total tokens processed count to $40,000,000$, which will be sufficient to produce reasonably fluent text. You may also increase the target validation loss from 1.45 to 2.00.

Running our solution code with a tuned learning rate on an M4 Max chip and 36 GB of RAM, we use batch size $\times$ total step count $\times$ context length $= 32 \times 5000 \times 256 = 40,960,000$ tokens, which takes 1 hour and 22 minutes on `cpu` and 36 minutes on `mps`. At step 5000, we achieve a validation loss of 1.80.

Some additional tips:

- When using $N$ training steps, we suggest adjusting the cosine learning rate decay schedule to terminate its decay (i.e., reach the minimum learning rate) at precisely step $N$.
- When using `mps`, do **not** use TF32 kernels, i.e., do **not** set

  ```
  torch.set_float32_matmul_precision('high')
  ```

  as you might with `cuda` devices. We tried enabling TF32 kernels with `mps` (`torch` version 2.9.0) and found the backend sometimes uses silently broken kernels that cause unstable training.

- You can speed up training by JIT-compiling your model with `torch.compile`. Specifically:
  - On `cpu`, compile your model with

    ```
    model = torch.compile(model)
    ```

  - On `mps`, you can somewhat optimize the backward pass using

    ```
    model = torch.compile(model, backend="aot_eager")
    ```

    Compilation with Inductor is not supported on `mps` as of `torch` version 2.9.0.

(b) Folk wisdom is that the best learning rate is "at the edge of stability." Investigate how the point at which learning rates diverge is related to your best learning rate.

**Deliverable**: Learning curves of increasing learning rate which include at least one divergent run and an analysis of how this relates to convergence rates.

Now let's vary the batch size and see what happens to training. Batch sizes are important – they let us get higher efficiency from our GPUs by doing larger matrix multiplies, but is it true that we always want batch sizes to be large? Let's run some experiments to find out.

**Problem (`batch_size_experiment`): Batch size variations (1 B200 hr) (1 point)**

---

Vary your batch size all the way from 1 to the GPU memory limit. Try at least a few batch sizes in between, including typical sizes like 64 and 128.

**Deliverable**: Learning curves for runs with different batch sizes. The learning rates should be optimized again if necessary.

**Deliverable**: A few sentences discussing your findings on batch sizes and their impacts on training.

With your decoder in hand, we can now generate text! We will generate from the model and see how good it is. As a reference, you should get outputs that look at least as good as the example below.

**Example (`ts_generate_example`): Sample output from a TinyStories language model**

---

Once upon a time, there was a pretty girl named Lily. She loved to eat gum, especially the big black one. One day, Lily's mom asked her to help cook dinner. Lily was so excited! She loved to help her mom. Lily's mom made a big pot of soup for dinner. Lily was so happy and said, "Thank you, Mommy! I love you." She helped her mom pour the soup into a big bowl. After dinner, Lily's mom made some yummy soup. Lily loved it! She said, "Thank you, Mommy! This soup is so yummy!" Her mom smiled and said, "I'm glad you like it, Lily." They finished cooking and continued to cook together. The end.

**Low-Resource Tip: Generate text on CPU or Apple Silicon**

---

If instead you used the low-resource configuration with 40M tokens processed, you should see generations that still resemble English but are not as fluent as above. For example, our sample output from a TinyStories language model trained on 40M tokens is below:

---

Once upon a time, there was a little girl named Sue. Sue had a tooth that she loved very much. It was his best head. One day, Sue went for a walk and met a ladybug! They became good friends and played on the path together.

"Hey, Polly! Let's go out!" said Tim. Sue looked at the sky and saw that it was difficult to find a way to dance shining. She smiled and agreed to help the talking!"

As Sue watched the sky moved, what it was. She

Here is the precise problem statement and what we ask for:

**Problem (`generate`): Generate text (1 point)**

---

Using your decoder and your trained checkpoint, report the text generated by your model. You may need to manipulate decoder parameters (temperature, top-p, etc.) to get fluent outputs.

**Deliverable**: Text dump of at least 256 tokens of text (or until the first `<|endoftext|>` token), and a brief comment on the fluency of this output and at least two factors which affect how good or bad this output is.

## 7.3 Ablations and architecture modification

The best way to understand the Transformer is to actually modify it and see how it behaves. We will now do a few simple ablations and modifications.

### Ablation 1: layer normalization

It is often said that layer normalization is important for the stability of Transformer training. But perhaps we want to live dangerously. Let's remove RMSNorm from each of our Transformer blocks and see what happens.

**Problem (`layer_norm_ablation`): Remove RMSNorm and train (0.5 B200 hrs) (1 point)**

---

Remove all of the RMSNorms from your Transformer and train. What happens at the previous optimal learning rate? Can you get stability by using a lower learning rate?

**Deliverable**: A learning curve for when you remove RMSNorms and train, as well as a learning curve for the best learning rate.

**Deliverable**: A few sentences of commentary on the impact of RMSNorm.

Let's now investigate another layer normalization choice that seems arbitrary at first glance. *Pre-norm* Transformer blocks are defined as

$$z = x + \text{MultiHeadSelfAttention}(\text{RMSNorm}(x)) \tag{25}$$

$$y = z + \text{FFN}(\text{RMSNorm}(z)). \tag{26}$$

This is one of the few 'consensus' modifications to the original Transformer architecture, which used a *post-norm* approach as

$$z = \mathrm{RMSNorm}(x + \mathrm{MultiHeadSelfAttention}(x)) \tag{27}$$
$$y = \mathrm{RMSNorm}(z + \mathrm{FFN}(z)). \tag{28}$$

Let's revert back to the *post-norm* approach and see what happens.

**Problem (`pre_norm_ablation`): Implement post-norm and train (0.5 B200 hrs) (1 point)**

---

Modify your pre-norm Transformer implementation into a post-norm one. Train with the post-norm model and see what happens.

**Deliverable**: A learning curve for a post-norm Transformer, compared to the pre-norm one.

We see that layer normalization has a major impact on the behavior of the Transformer, and that even the position of the layer normalization is important.

### Ablation 2: position embeddings

We will next investigate the impact of the position embeddings on the performance of the model. Specifically, we will compare our base model (with RoPE) with not including position embeddings at all (NoPE). It turns out that decoder-only transformers, i.e., those with a causal mask as we have implemented, can in theory infer relative or absolute position information without being provided with position embeddings explicitly [Y.-H. H. Tsai et al., 2019; A. Kazemnejad et al., 2023]. We will now test empirically how NoPE performs compared to RoPE.

**Problem (`no_pos_emb`): Implement NoPE (0.5 B200 hrs) (1 point)**

---

Modify your Transformer implementation with RoPE to remove the position embedding information entirely, and see what happens.

**Deliverable**: A learning curve comparing the performance of RoPE and NoPE.

### Ablation 3: SwiGLU vs. SiLU

Next, we will follow N. Shazeer [20] and test the importance of gating in the feed-forward network, by comparing the performance of SwiGLU feed-forward networks versus feed-forward networks using SiLU activations but no gated linear unit (GLU):

$$\mathrm{FFN}_{\mathrm{SiLU}}(x) = W_2\ \mathrm{SiLU}(W_1 x). \tag{29}$$

Recall that in our SwiGLU implementation, we set the dimensionality of the inner feed-forward layer to be roughly $d_{\mathrm{ff}} = \frac{8}{3} d_{\mathrm{model}}$ (while ensuring that $d_{\mathrm{ff}} \bmod 64 = 0$, to make use of GPU tensor cores). In this ablation baseline, your $\mathrm{FFN}_{\mathrm{SiLU}}$ implementation should instead set $d_{\mathrm{ff}} = 4 \times d_{\mathrm{model}}$, to approximately match the parameter count of the default SwiGLU feed-forward network (which has three instead of two weight matrices).

**Problem (`swiglu_ablation`): SwiGLU vs. SiLU (0.5 B200 hrs) (1 point)**

---

**Deliverable**: A learning curve comparing the performance of SwiGLU and SiLU feed-forward networks, with approximately matched parameter counts.

**Deliverable**: A few sentences discussing your findings.

**Low-Resource Tip: Online students with limited GPU resources should test modifications on TinyStories**

---

In the remainder of the assignment, we will move to a larger-scale, noisier web dataset (OpenWebText), experimenting with architecture modifications and (optionally) making a submission to the course leaderboard.

It takes a long time to train an LM to fluency on OpenWebText, so we suggest that online students with limited GPU access continue testing modifications on TinyStories (using validation loss as a metric to evaluate performance).

## 7.4 Running on OpenWebText

We will now move to a more standard pretraining dataset created from a web crawl. A small sample of OpenWebText [A. Gokaslan et al., 2019] is also provided as a single text file: see Section 1 for how to access this file.

Here is an example from OpenWebText. Note how the text is much more realistic, complex, and varied. You may want to look through the training dataset to get a sense of what training data looks like for a web-scraped corpus.

**Example (`owt_example`): One example from OWT**

---

Baseball Prospectus director of technology Harry Pavlidis took a risk when he hired Jonathan Judge.

Pavlidis knew that, as Alan Schwarz wrote in The Numbers Game, "no corner of American culture is more precisely counted, more passionately quantified, than performances of baseball players." With a few clicks here and there, you can find out that Noah Syndergaard's fastball revolves more than 2,100 times per minute on its way to the plate, that Nelson Cruz had the game's highest average exit velocity among qualified hitters in 2016 and myriad other tidbits that seem ripped from a video game or science fiction novel. The rising ocean of data has empowered an increasingly important actor in baseball's culture: the analytical hobbyist.

That empowerment comes with added scrutiny – on the measurements, but also on the people and publications behind them. With Baseball Prospectus, Pavlidis knew all about the backlash that accompanies quantitative imperfection. He also knew the site's catching metrics needed to be reworked, and that it would take a learned mind – someone who could tackle complex statistical modeling problems – to complete the job.

"He freaks us out." Harry Pavlidis

Pavlidis had a hunch that Judge "got it" based on the latter's writing and their interaction at a site-sponsored ballpark event. [...]

**Note:** You may have to re-tune your hyperparameters such as learning rate or batch size for this experiment.

**Problem (`main_experiment`): Experiment on OWT (2 B200 hrs) (2 points)**

---

Train your language model on OpenWebText with the same model architecture and total training iterations as TinyStories. How well does this model do?

**Deliverable**: A learning curve of your language model on OpenWebText. Describe the difference in losses from TinyStories – how should we interpret these losses?

**Deliverable**: Generated text from OpenWebText LM, in the same format as the TinyStories outputs. How is the fluency of this text? Why is the output quality worse even though we have the same model and compute budget as TinyStories?

## 7.5 Your own modification + leaderboard

Congratulations on getting to this point. You're almost done! You will now try to improve upon the Transformer architecture, and see how your hyperparameters and architecture stack up against other students in the class.

### Rules for the leaderboard

There are no restrictions other than the following:

**Runtime:** Your submission can run for at most 45 minutes on a B200. You might want to enforce this in your submission script if you use either SLURM or Modal.

**Data:** You may only use the OpenWebText training dataset that we provide.

Otherwise, you are free to do whatever your heart desires.

If you are looking for some ideas on what to implement, you can check out some of these resources:

- State-of-the-art open-source LLM families, such as Llama 3 [A. Grattafiori et al., 2024] or Qwen 2.5 [A. Yang et al., 2024].
- The NanoGPT speedrun repository (github.com/KellerJordan/modded-nanogpt), where community members post many interesting modifications for "speedrunning" small-scale language model pretraining. For example, a common modification that dates back to the original Transformer paper is to tie the weights of the input and output embeddings together (see A. Vaswani et al. [8] (Section 3.4) and A. Chowdhery et al. [16] (Section 2)). If you do try weight tying, you may have to decrease the standard deviation of the embedding/LM head init.

You will want to test these on either a small subset of OpenWebText or on TinyStories before trying the full 45-minute run.

As a caveat, we do note that some of the modifications you may find working well in this leaderboard may not generalize to larger-scale pretraining. We will explore this idea further in the scaling laws unit of the course.

**Problem (`leaderboard`): Leaderboard (10 B200 hrs) (6 points)**

---

You will train a model under the leaderboard rules above with the goal of minimizing the validation loss of your language model within 0.75 B200-hours.

**Deliverable**: The final validation loss that was recorded, an associated learning curve that clearly shows a wall-clock-time x-axis that is less than 45 minutes, and a description of what you did. We expect a leaderboard submission to beat at least the naive baseline of a 5.0 loss. Submit to the leaderboard here: github.com/stanford-cs336/assignment1-basics-leaderboard.

## Bibliography


[1] R. Eldan and Y. Li, “TinyStories: How Small Can Language Models Be and Still Speak Coherent English?.” 2023.

[2] A. Gokaslan, V. Cohen, E. Pavlick, and S. Tellex, “OpenWebText corpus.” 2019.

[3] R. Sennrich, B. Haddow, and A. Birch, “Neural Machine Translation of Rare Words with Subword Units,” in *Proc. of ACL*, 2016.

[4] C. Wang, K. Cho, and J. Gu, “Neural Machine Translation with Byte-Level Subwords.” 2019.

[5] P. Gage, “A new algorithm for data compression,” *C Users Journal*, vol. 12, no. 2, pp. 23–38, Feb. 1994.

[6] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language Models are Unsupervised Multitask Learners.” 2019.

[7] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, “Improving Language Understanding by Generative Pre-Training.” 2018.

[8] A. Vaswani *et al.*, “Attention is All you Need,” in *Proc. of NeurIPS*, 2017.

[9] T. Q. Nguyen and J. Salazar, “Transformers without Tears: Improving the Normalization of Self-Attention,” in *Proc. of IWSWLT*, 2019.

[10] R. Xiong *et al.*, “On Layer Normalization in the Transformer Architecture,” in *Proc. of ICML*, 2020.

[11] J. L. Ba, J. R. Kiros, and G. E. Hinton, “Layer Normalization.” 2016.

[12] H. Touvron *et al.*, “LLaMA: Open and Efficient Foundation Language Models.” 2023.

[13] B. Zhang and R. Sennrich, “Root Mean Square Layer Normalization,” in *Proc. of NeurIPS*, 2019.

[14] A. Grattafiori *et al.*, “The Llama 3 Herd of Models.” [Online]. Available: https://arxiv.org/abs/2407.21783

[15] A. Yang *et al.*, “Qwen2.5 Technical Report,” *arXiv preprint arXiv:2412.15115*, 2024.

[16] A. Chowdhery *et al.*, “PaLM: Scaling Language Modeling with Pathways.” 2022.

[17] D. Hendrycks and K. Gimpel, “Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units.” 2016.

[18] S. Elfwing, E. Uchibe, and K. Doya, “Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning.” [Online]. Available: https://arxiv.org/abs/1702.03118

[19] Y. N. Dauphin, A. Fan, M. Auli, and D. Grangier, “Language Modeling with Gated Convolutional Networks.” [Online]. Available: https://arxiv.org/abs/1612.08083

[20] N. Shazeer, “GLU Variants Improve Transformer.” 2020.

[21] J. Su, Y. Lu, S. Pan, B. Wen, and Y. Liu, “RoFormer: Enhanced Transformer with Rotary Position Embedding.” 2021.

[22] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” in *Proc. of ICLR*, 2015.

[23] I. Loshchilov and F. Hutter, “Decoupled Weight Decay Regularization,” in *Proc. of ICLR*, 2019.

[24] T. B. Brown *et al.*, “Language Models are Few-Shot Learners,” in *Proc. of NeurIPS*, 2020.

[25] J. Kaplan *et al.*, “Scaling Laws for Neural Language Models.” 2020.

[26] J. Hoffmann *et al.*, “Training Compute-Optimal Large Language Models.” 2022.

[27] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, “The Curious Case of Neural Text Degeneration,” in *Proc. of ICLR*, 2020.

[28] Y.-H. H. Tsai, S. Bai, M. Yamada, L.-P. Morency, and R. Salakhutdinov, “Transformer Dissection: An Unified Understanding for Transformer`s Attention via the Lens of Kernel,” in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, K. Inui, J. Jiang, V. Ng, and X. Wan, Eds., Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 4344–4353. doi: 10.18653/v1/D19-1443.

[29] A. Kazemnejad, I. Padhi, K. Natesan, P. Das, and S. Reddy, “The Impact of Positional Encoding on Length Generalization in Transformers,” in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: https://openreview.net/forum?id=Drrl2gcjzl